CHAPTER 26

Create Classes That Modify Their Subclasses

One of the fundamental principles of programming goes something like this: Never leave to a human that which can be done by a program. Or, to put it another way, some of the greatest leaps in software engineering have happened because some lazy sod simply got tired of repeating steps two through six over and over. So far in our adventures with Ruby's open classes we have seen how we can change classes with monkey patching, a more or less manual process of slapping some new code over the existing code. From there we moved to building classes that could change themselves, classes that say "Gee, I'm running in Ruby 1.8, so I had better define this method." In this chapter we'll make the final leap: We'll move that class-modifying code out of the class being modified and up into a superclass. We'll see that by doing this we can dramatically increase the amount of metaprogramming leverage that we can apply. With a bit of luck, this technique will allow you to finish your programs that much quicker and go back to your favorite pastime, being a lazy sod.

A Document of Paragraphs

Thus far we have kept our `Document` class conveniently, but unrealistically, simple. The sad fact is that you can't model the content of real-world documents with a simple string. Real documents have paragraphs and fonts, fonts that come in flavors like normal and bold and italics. A more realistic document class might actually start with a class for paragraphs:

```
class Paragraph
  attr_accessor :font_name, :font_size, :font_emphasis
  attr_accessor :text

  def initialize( font_name, font_size, font_emphasis, text='')
    @font_name = font_name
    @font_size = font_size
    @font_emphasis = font_emphasis
    @text = text
  end

  def to_s
    @text
  end

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

And only then turn to the document itself:

```
class StructuredDocument
  attr_accessor :title, :author, :paragraphs

  def initialize( title, author )
    @title = title
    @author = author
    @paragraphs = []
    yield( self ) if block_given?
  end

  def <<( paragraph )
    @paragraphs << paragraph
  end

  def content
    @paragraphs.inject('') { |text, para| "#{text}\n#{para}" }
  end

  # ...
end
```

The new `StructuredDocument` class is mostly just a collection of paragraphs, where each paragraph consists of some text, a font name (i.e., `:arial`), a font size (perhaps 12 point), and an emphasis, something like `:bold` or `:italic`. Thus we might create a resume like this:[1]

```
russ_cv = StructuredDocument.new( 'Resume', 'RO' ) do |cv|
  cv << Paragraph.new( :nimbus, 14, :bold, 'Russ Olsen' )
  cv << Paragraph.new( :nimbus, 12, :italic, '1313 Mocking Bird Lane')
  cv << Paragraph.new( :nimbus, 12, :none, 'russ@russolsen.com')
  # .. and so on
end
```

Armed with the newfound power of the `StructuredDocument` class, we can build all sorts of specialized documents, everything from resumes to instructions for installing an LCD TV.

Subclassing Is (Sometimes) Hard to Do

The trouble is, all that building is going to be real work and, because most resumes and instruction manuals look more or less alike, fairly repetitious work at that. If our users are creating a lot of documents, we might want to help them along. Easy enough. We simply cook up a number of subclasses, each with some helpful methods. We might, for instance, create a subclass for resumes:

```
class Resume < StructuredDocument
  def name( text )
    paragraph = Paragraph.new( :nimbus, 14, :bold, text )
    self << paragraph
  end

  def address( text )
    paragraph = Paragraph.new( :nimbus, 12, :italic, text )
    self << paragraph
  end
```

1. Note the clever use of the initialize block!

```
    def email( text )
      paragraph = Paragraph.new( :nimbus, 12, :none, text )
      self << paragraph
    end

    # and so on
  end
```

Using these methods, you can programmically build a resume with a minimum of fuss:

```
  russ_cv = Resume.new( 'russ', 'resume') do |cv|
    cv.name( 'Russ Olsen' )
    cv.address( '1313 Mocking Bird Lane' )
    cv.email( 'russ@russolsen.com' )

    # Etc...
  end
```

You might also do something similar with installation instructions:

```
  class Instructions < StructuredDocument
    def introduction( text )
      paragraph = Paragraph.new( :mono, 14, :none, text )
      self << paragraph
    end

    def warning( text )
      paragraph = Paragraph.new( :arial, 22, :bold, text )
      self << paragraph
    end

    def step( text )
      paragraph = Paragraph.new( :nimbus, 14, :none, text )
      self << paragraph
    end

    # and so on
  end
```

If we step back and look at our handiwork so far, we can see that it's a bit of a mixed bag. On the plus side, we have built a couple of friendly, user-oriented classes. If you need to build a resume or a set of instructions, we have a handy class that will help you out. The problem is that we still have a lot of repetitive code. Every one of our helper methods looks almost exactly like every other helper method. The method that adds an e-mail address to a resume is, plus or minus a font name and size, identical to the one that adds a warning to a set of instructions. Sadly, there seems to be no cure for this: If we want all those nice helper methods, we simply need to sit down and build them, right?

Class Methods That Build Instance Methods

Perhaps not. Armed with the knowledge that (a) you can change any Ruby class at any time, and (b) Ruby classes definitions are executed, we might just be able to avoid all of this redundant code. The effect we're looking for is to be able to say, "The `Instruction` class needs to have a method called `introduction` that will add a new paragraph rendered in the italic version of Arial at a glorious 18 points" and have Ruby create the method for you. Something like this:

```
class Instructions < StructuredDocument
  paragraph_type( :introduction,
    :font_name => :arial,
    :font_size => 18,
    :font_emphasis => :italic )

  # And so on...
end
```

Let's try to make `paragraph_type` a reality, one step at a time. First, since we are calling it right inside the class definition, it's clear that `paragraph_type` needs to be a *class* method:

```
class StructuredDocument
  def self.paragraph_type( paragraph_name, options )
    # What do we do in here?
  end

  # ...
end
```

The `paragraph_type` class method takes two arguments: the name of the new paragraph type (which is also going to be the name of the method we're going to add to the class) and a hash of options, one that will contain things like the font name and size. The real question is, what do we do inside of the `paragraph_type` class method? The answer is that we need to define a new instance method on the subclass of `StructuredDocument` that is calling the `paragraph_type` method. Your natural impulse is to write something like this:

```
class StructuredDocument

  def self.paragraph_type( paragraph_name, options )
    def <<the new method>>
        # Add a new paragraph
    end
  end

  # ...
end
```

Unfortunately that is not going to work. The trouble with the familiar `def` statement is that we have the name of the method we're trying to create in the `name` parameter but `def` requires an explicit "you type it right here" name for the method. In other words, we have the name of the new method as *data*, but `def` wants it as *code*.

The way around this roadblock is to make use of `class_eval`. Built into all Ruby classes, the `class_eval` method takes a string and evaluates it as if it were code that appeared in the class body. This is exactly what we need: We'll just build a string that contains the code for the new method definition and then `class_eval` the new method into reality:

```
class StructuredDocument
  def self.paragraph_type( paragraph_name, options )

    name = options[:font_name] || :arial
    size = options[:font_size] || 12
    emphasis = options[:font_emphasis] || :normal

    code = %Q{
      def #{paragraph_name}(text)
```

```
          p = Paragraph.new(:#{name}, #{size}, :#{emphasis}, text)
          self << p
        end
      }
      class_eval( code )
    end

    # ...
  end
```

To step through the code shown, imagine that we have this call to `paragraph_type`:

```
  class Instructions < StructuredDocument
    paragraph_type( :introduction,
      :font_name => :arial,
      :font_size => 18,
      :font_emphasis => :italic )

    # And so on...
  end
```

The `paragraph_type` method starts by pulling the font name, size, and emphasis out of the `options` hash, filling in defaults as needed. Now for the interesting part: The `paragraph_type` method creates a string that contains the code for the new `introduction` instance method, a string that will look something like this:

```
    def introduction(text)
      p = Paragraph.new(:arial, 18, :italics, text)
      self << p
    end
```

Finally, `paragraph_type` uses `class_eval` to execute the string, which creates the `introduction` method. Note that the new method ends up on the `StructuredDocument` subclass (i.e., `Instructions`) and not on the `StructuredDocument` class itself because the whole process started with a call from inside the `Instructions` class, which set `self` to `Instructions`.

Better Method Creation with define_method

Although creating new methods by `class_eval`'ing a string has a certain clarity to it—if nothing else, you can print out the string and actually see the method that you're defining—it is really not ideal. We generally like to avoid "evaluate some code on the fly"-type methods if there is a more normal API alternative. In the case of defining a new method, there definitely is an alternative, one with the very obvious name of `define_method`. To use `define_method`, you call it with the name of the new method and a block. You end up with a new method with the given name that will execute the block when called. Conveniently, the parameters of the block become the parameters of the new method. Thus, we can reformulate `paragraph_type` using `define_method` very easily:

```
class StructuredDocument
  def self.paragraph_type( paragraph_name, options )
    name = options[:font_name] || :arial
    size = options[:font_size] || 12
    emphasis = options[:font_emphasis] || :none

    define_method(paragraph_name) do |text|
      paragraph = Paragraph.new( name, size, emphasis, text )
      self << paragraph
    end
  end

  # ...
end
```

Either way you do it, you end up with a method in the superclass that can add methods to its subclasses. You also have a very powerful technique for creating custom subclasses with very little effort.

The Modification Sky Is the Limit

So far we have only been talking about writing classes that can hang new methods on their subclasses. Once you have the basic idea down, however, there is no limit to what you can do to a subclass from a superclass method. You might, for example, add a method that changes the visibility of the document methods in a subclass:

```
class StructuredDocument

  # Rest of the class omitted...

  def self.privatize
    private :content
  end
end
```

The thing to note about this code is that it doesn't change the accessibility of the `content` method on all `StructuredDocument` instances. But it does give its subclasses an easy way of declaring the content method private:

```
class BankStatement < StructuredDocument
  paragraph_type( :bad_news,
    :font_name => :arial,
    :font_size => 60,
    :font_emphasis => :bold )

  privatize
end
```

Now if you try to get at the contents of a `BankStatement`:

```
statement = BankStatement.new( 'Bank Statement', 'Russ')
statement.bad_news("You're broke!")
puts statement.content
```

You are in for a shock:

```
private method `content' called for #<BankStatement:0x8b8f544>
```

Nor are you limited to just messing with instance methods. You might, for example, want to mark any document that you privatize with a class method to announce that the document is confidential:

```
class StructuredDocument

  # Rest of the class omitted...
  def self.disclaimer
    "This document is here for all to see"
  end

  def self.privatize
    private :content

    def self.disclaimer
      "This document is a deep, dark secret"
    end
  end
end
```

This code starts by defining a default `disclaimer` method on the `StructuredDocument` class, a method that proclaims the availability of the document. If you create a `StructuredDocument` subclass and don't call `privatize`, this is the method you will get. If, on the other hand, you do call `privatize` from your subclass:

```
class BankStatement < StructuredDocument
  paragraph_type( :bad_news,
    :font_name => :arial,
    :font_size => 60,
    :font_emphasis => :bold )

  privatize
end
```

Then `BankStatement` will end up with its own version of the disclaimer method, so that running:

```
puts BankStatement.disclaimer
```

Will give you this:

```
This document is a deep, dark secret
```

Remember, if you can do it to a class, you can do it from the superclass!

In the Wild

Once you know what to look for, it's hard to miss Ruby subclass-changing methods in real-world code. In fact, some are built into every Ruby class that you've ever written. These ubiquitous methods go by the name `attr_accessor`, `attr_reader`, and `attr_writer`. Think about what `attr_accessor` and friends do: They let you describe some methods that you want, so that if you say this:

```
class Printer
  attr_accessor :name
end
```

You really end up with a class that looks something like this:

```
class Printer
  def name
    @name
  end

  def name=(value)
    @name = value
  end
end
```

If you have read this far into this chapter, figuring out how `attr_accessor` and friends works is not hard. Here's a simple version of `attr_reader`[2] that uses `class_eval`:

```
class Object
  def self.simple_attr_reader(name)
    code = "def #{name}; @#{name}; end"
    class_eval( code )
  end
end
```

2. The real `attr_reader` method, along with its siblings, is defined in the Ruby C code (or Java if you are using JRuby). If you think that this makes the implementations of those methods inaccessible, then you need to read Chapter 30.

Similarly, here's a stripped-down version of `attr_writer`, this time using `define_method`:

```
class Object
  def self.simple_attr_writer(name)
    method_name = "#{name}="
    define_method( method_name ) do |value|
      variable_name = "@#{name}"
      instance_variable_set( variable_name, value )
    end
  end
end
```

Aside from `attr_accessor` et al., the most well-known subclass changing methods are probably those that come with ActiveRecord. As every Rails programmer knows, doing this:

```
class Automobile > ActiveRecord::Base
  has_one :manufacturer
end

my_car  = Automobile.find( :first )
```

Means that you can say `my_car.manufacturer` to get to the object that represents the company that made the car.

Finally, if you look in `forwardable.rb`, which is part of the Ruby standard library, you will find still more examples of class-modifying methods. The idea behind `forwardable.rb` is similar to `delegate.rb`, which we met when we were talking about using `method_missing` to delegate. Both of these library classes try to make delegation easy. The difference is that where `delegate.rb` takes the `method_missing` approach to delegation, catching calls to nonexistent methods and sending them off to the other object, `Forwardable` actually generates the delegating methods on the fly.

The interesting thing about `Forwardable` is that it's not a class at all. It's a module that you mix in at the class level with `extend`. Other than its slightly surprising packaging, `Forwardable` works exactly like the examples in this chapter. Here, for instance, is a simple class that looks like a document, but that actually just bounces all the method calls to `title`, `author`, and `content` off to a real `Document` instance:

```
class DocumentWrapper
  extend Forwardable

  def_delegators  :@real_doc, :title, :author, :content

  def initialize( real_doc )
    @real_doc = real_doc
  end
end
```

Like our earlier `delegate.rb`-based `DocumentWrapper`, this one lets you treat the wrapper just like the original:

```
real_doc = Document.new( 'Two Cities', 'Dickens', 'It was...' )
wrapped_doc = DocumentWrapper.new( real_doc )

puts wrapped_doc.title
puts wrapped_doc.author
puts wrapped_doc.content
```

Here is an edited version of the key method in `Forwardable`:

```
module Forwardable
  # Lots of code deleted...

  def def_instance_delegator(accessor, method, ali = method)
    str = %{
      def #{ali}(*args, &block)
        #{accessor}.__send__(:#{method}, *args, &block)
      end
    }
    module_eval(str, __FILE__, line_no)
  end
end
```

Aside from the fact that this code uses `module_eval`—which is synonymous with `class_eval`—this is exactly the string-based technique that we've been discussing.

Staying Out of Trouble

It's easy to get lost when you first try to do the sort of metaprogramming we looked at in this chapter. The problem is not that this is an extremely complex process; it's just that most programmers new to Ruby are not used to thinking in terms of modifying classes at all, let alone writing methods to do the modification for them, let alone moving those methods up to a superclass. If you feel like this technique has left you in a confusing maze of twisty passages,[3] here are some signposts that can help.

First, keep your goal firmly in mind. In our example, we wanted to make it easy to add paragraph-generating methods to `StructuredDocument` subclasses.

Second, know when things happen. For example, when you load the `StructuredDocument` code, perhaps with this:

```
require 'structured_document'
```

You end up with the generic `StructuredDocument` class, which has the `paragraph_type` class method on it. A bit later (perhaps even on the very next line of code) you load the `Instructions` class:

```
require 'instructions'
```

As the `Instructions` class definition is getting executed, it will fire off calls to the `paragraph_type` method up in the `StructuredDocument` class. This will add methods with names like `introduction`, `warning`, and `step` to the `Instructions` class. Only when all of this defining is over do you make an instance of `Instructions` and call the generated methods.

The third thing to know is that the value of `self` is at every stage of this process. This starts out relatively straightforward but gets a little hairy as you go along. The straightforward bit is in the superclass:

```
class StructuredDocument

  # Self is StructuredDocument here

  def self.paragraph_type( paragraph_name, options )
    # ...
```

3. All of them alike.

```
    end

  end
```

When you say `def self.paragraph_type`, you are defining a new class method on the `StructuredDocument` class. But—and this is a key "but"—when you call the `paragraph_type` method from the subclass, `self` will be whatever class called the method. Thus, `self` inside `paragraph_type` will be `Resume` or `Instructions` (the subclass), not `StructuredDocument`. This is why the `introduction` method ends up on `Instructions` and not on `StructuredDocument`.

Finally, when you actually call the generated method on an instance of your subclass, maybe like this:

```
  omlette_howto = Instructions.new( 'Russ', 'Omlettes' )  do |i|
    i.warning( "Careful of those sharp egg shells!")
  end
```

The value of `self` inside the generated method will be the `Instructions` instance. As the philosophers say, "Know thyself"!

There is one other thing to beware of with the metaprogramming techniques we've covered in the past few chapters—the inclination of programmers to either avoid them completely or use them for everything in sight. So the question is: When should you pull out the metaprogramming and when should you stick to garden-variety code?

As usual, the extreme cases are the easiest. If you can solve your problem in a reasonable way with your traditional programming chops, do that. Do you need to share the same method between several different classes? You *could* add that method on the fly to both classes, but probably you are better off putting the method in a superclass or a mixin module. In the quest for software that is actually useful, simple solutions to straightforward problems have a plain eloquence all their own.

At the other extreme are the things you just cannot do without metaprogramming. If you need to build a generic method-agnostic proxy or a reloadable `Document` class, metaprogramming is the only solution.

The intermediate cases, problems like our "to encrypt or not to encrypt?" question and the Ruby 1.8/1.9 incompatibility handler are harder. If you can solve a problem with some traditional code or a touch of metaprogramming, how do you decide?

Again, it's all about striking a balance. Metaprogramming can dramatically reduce the volume and the complexity of your code. Hooks let you define code that will run at helpful times. Method missing and the techniques that rely on open classes can save you an enormous amount of coding toil. All of this, however, comes at a price. Those hooks might go off at unexpected times. Using method missing and open classes means that your application will be running code that has no obvious counterpart in your source tree.

The trick is to simply get back more programming goodness than you pay in meta-complexity. Take the Ruby 1.8/1.9 string example that we looked at in the last chapter. In real life, would I actually use class-level logic to deal with that one problem? Probably not. In real life I would probably code some ordinary run-time logic that asked itself about the Ruby version and did the right thing. If, however, I was doing this with scores of methods, or if the *this* or *that* logic was complex, I might well turn to metaprogramming. The bottom line is simple: Make that metaprogramming pay for itself.

Wrapping Up

The trouble with traditional programming languages is that they treat classes as if they were petrified. You make a class and there it is, eternal and unchanging. As we have seen in the last few chapters, the Ruby way of looking at things is that classes are objects just like any other. If you are somehow unhappy with the contents of your class, you can change them. Two chapters ago we saw how you can change Ruby classes manually, via monkey patching. In the last chapter we saw how you can write classes that reconfigure themselves. Finally, in this chapter we saw how you can share all of that class-changing logic by moving it into a superclass or a module.

One of the advantages of metaprogramming is the way you can use it to give your Ruby a customized "made just for this problem" feel. Page back and look at the examples in this chapter. One thing that stands out is how the `paragraph_type` and `privatize methods` feel less like methods and more like key words, specialized parts of a new Ruby-like language. Now there's an idea worth pursuing.