CHAPTER 16
Use Modules as Mixins

I started off the last chapter by pointing out that a class is a combination of two things, a container and a factory. We build classes full of code (that's the container part) and then we use them to manufacture instances. One thing that I glossed over in the last chapter is that along with being containers and factories, Ruby classes can also be *super*: Like classes in most other object oriented programming languages, Ruby classes are arranged in an inheritance tree, so that a key part of constructing a new class is picking its parent, or superclass.

In this chapter we are going to discover that you can also insert, or "mix in," modules into the inheritance tree of your classes. If you haven't come across the idea of a mixin before, let me whet your appetite. Mixins allow you to easily share common code among otherwise unrelated classes. Mixins are custom-designed for those situations where you have a method or six that need to be included in a number of different classes that have nothing else in common. During our tour of mixin modules we'll have a good look at how they work and at how real some real Ruby applications like Rails use mixins.

Better Books with Modules

Let's begin our look at mixins by continuing our quest to improve—or at least measure—the quality of prose entrusted to our `Document` class. Recall that we've already written a method to measure the average length of the words in our documents. There is, however, more to good writing than modestly sized words. Editors also tend to

frown on clichés, those phrases that have been so overused that they've lost all their rhetorical pizzazz. We decide that an array of regular expressions along with a new `Document` method[1] will do the trick. So we are off and running:

```
class  Document

  CLICHES = [ /play fast and loose/,
              /make no mistake/,
              /does the trick/,
              /off and running/,
              /my way or the highway/ ]

  def number_of_cliches
     CLICHES.inject(0) do |count, phrase|
       count += 1 if phrase =~ content
       count
     end
  end

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

This `number_of_cliches` method is like much of the code we write—not terribly exciting, but useful nevertheless. So, you release your new `Document` class and everyone is happy. Unfortunately—at least for you—your employers decide that there might be a world beyond traditional paper books and begin to move into the digital market. To this end, they buy an eBook publishing system, which features its own class hierarchy for representing books:

```
class ElectronicBook < ElectronicText
   # Lots of complicated stuff omitted...
end
```

This is a problem because everyone loves the `number_of_cliches` method so much that they would like to get it into the new `ElectronicBook` class. So how do you insert a method into these two, very separate class hierarchies?

1. Again, we are starting over with our original `Document` class.

Your gut reaction—at least if your gut went to the same object oriented programming school as mine—might be to create some sort of common superclass for both `ElectronicBook` and `Document`, maybe `Tome`. This superclass would contain the `number_of_cliches` method, which would then be neatly inherited by both `ElectronicBook` and `Document`. Sensible, but not always doable. Sometimes, especially in cases where you have inherited large bodies of code, it's just not practical to rewire the basic structure of your classes in order to share a few tens of lines of code.[2]

Mixin Modules to the Rescue

The way to solve the problem of sharing code among otherwise unrelated classes is by creating a mixin module. Let's take this step by step: First, you move the `number_of_cliches` method into a module:

```
module WritingQuality

  CLICHES = [ /play fast and loose/,
              /make no mistake/,
              /does the trick/,
              /off and running/,
              /my way or the highway/ ]

  def number_of_cliches
    CLICHES.inject(0) do |count, phrase|
      count += 1 if phrase =~ content
      count
    end
  end
end
```

Note that the `number_of_cliches` method is an ordinary *instance* method in the module, *not* a module-level method. Next, you include the module into the classes that need the method:

2. The astute reader will notice that I have passed silently over the possibility of simply copying the code into both classes. We shall speak no more of such an unpleasant alternative.

```
class Document
  include WritingQuality

  # Lots of stuff omitted...
end

class ElectronicBook < ElectronicText
  include WritingQuality

  # Lots of stuff omitted...
end
```

And you are done. When you include a module into a class, the module's methods magically become available to the including class, which means that you can now say this:

```
text = "my way or the highway does the trick"
my_tome = Document.new('Hackneyed', 'Russ', text)
puts my_tome.number_of_cliches
```

As well as this:

```
my_ebook = ElectronicBook.new( 'EHackneyed', 'Russ', text)
puts my_ebook.number_of_cliches
```

The Ruby jargon is that by including a module in a class you have **mixed it in** to the class. We say that the module itself, `WritingQuality` in this case, is a **mixin module.** A very useful aspect of mixins is that they are not limited by the "one superclass is all you get" rule. You can mix as many modules into a class as you like. For example, if you created `ProjectManagement` and `AuthorAccountTracking` modules for your publishing employer, you could include them both in the `ElectronicBook` class, along with `WritingQuality`:

```
module ProjectManagement
  # Lots of boring stuff omitted
end
```

```
module AuthorAccountTracking
  # Lots of even more boring stuff omitted
end

class  ElectronicBook < ElectronicText
  include WritingQuality
  include ProjectManagement
  include AuthorAccountTracking

  # Lots of stuff omitted...
end
```

In practice, this means that if you have several unrelated classes that need to share some code, you don't have to resort to restructuring your whole inheritance tree to get at that code. All you need to do is wrap the common stuff in a module and include that module in the classes that need it.

Extending a Module

Sometimes it's the class itself—as opposed to the instances of the class—that needs help from a module. Sometimes you want to pull in a module so that all the methods in the module become *class* methods. You might, for instance, have a module full of methods that know how to locate documents:

```
module Finders
  def find_by_name( name )
    # Find a document by name...
  end

  def find_by_id( doc_id )
    # Find a document by id
  end
end
```

You would like to get the `Finders` methods into your `Document` class as *class* methods. One way to get this done is to do this:

```
class Document
  # Most of the class omitted...
```

```
  class << self
    include Finders
  end
end
```

This code includes the module into the singleton class of `Document`, effectively making the methods of `Finders` singleton—and therefore class—methods of `Document`:

```
war_and_peace = Document.find_by_name( 'War And Peace' )
```

Including modules into the singleton class is a common enough task that Ruby has a special shortcut for it in the form of `extend`:

```
class Document
  extend Finders

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

Run this code and you will end up with class-level `Document.find_by_name` and `find_by_id` methods.

Staying Out of Trouble

Clearly there is something magical going on when you include a module in a class. The key bit of legerdemain is this: When you mix a module into a class, Ruby rewires the class hierarchy a bit, inserting the module as a sort of pseudo superclass of the class. As shown in Figure 16-1, the module gets interposed between the class and its original superclass.

This explains how the module methods appear in the including classes—they effectively become methods just up the inheritance chain from the class.

Although including a module inserts it into the class hierarchy, Ruby is a bit circumspect about this fact: No matter how many modules a class includes, instances of the class will still claim to be, well, instances of the class. So, if `my_tome` is an instance of `ElectronicBook`, then `my_tome.class` will return `ElectronicBook` no matter how many modules the `ElectronicBook` class includes. Module inclusion is not a complete

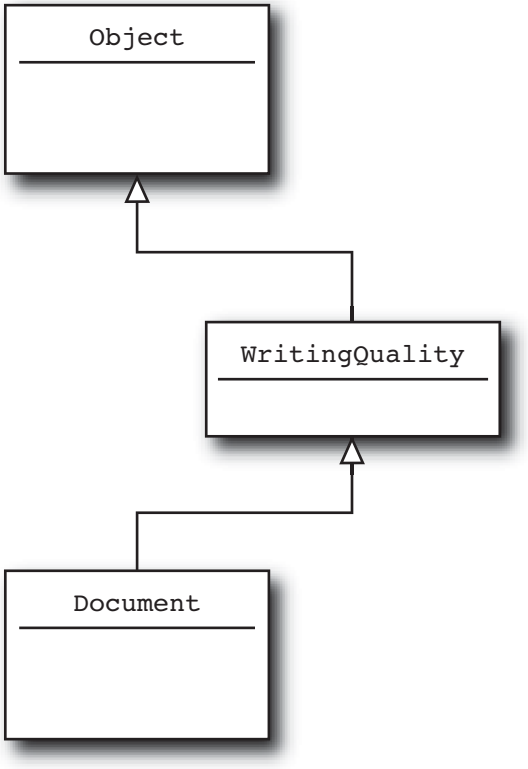

Figure 16-1 A module becomes a pseudo superclass to the class that includes it.

secret, however. You can discover whether the class of an instance includes a given module with the `kind_of?` method. So if `Document` includes the `WritingQuality` module, then `my_tome.kind_of?(WritingQuality)` will return true. You can also use the `ancestors` method to see the complete inheritance ancestry—modules included—of a class, so that `Document.ancestors` might return the following array:

```
[Document, WritingQuality, Object, Kernel, BasicObject]
```

This "insert the module in before the superclass" mechanism is not just some implementation detail that's liable to change at any moment. The way modules work is an integral part of the way that Ruby works. Nor is all of this purely academic—the module inclusion mechanism has some real, practical implications. For example, imagine that your company gets into the business of publishing political tracts such as

congressional speeches and the Governor's state of the state statement. Now, where politics are involved the rules of good writing change: Politics is where the clichés go when they die. Given this, you decide that there really is no point in reporting back on the number of clichés in political writing:

```
class Document
  include WritingQuality

  # Rest of the class omitted...
end

class PoliticalBook < Document
  def number_of_cliches
    0
  end

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

This seems like a sensible way to ensure that the `number_of_cliches` method always returns zero, but will it actually work? To put it another way, can you override a method in a module by defining that method in the class that includes the module?

Once you know that a mixin module effectively becomes a superclass when it is included, the answer is easy to come by: Yes. Since we know that the methods in a superclass cannot override the methods in subclasses, we can deduce that no module method can ever override a method in its host class. Call the method and Ruby will look first in the class, find it there, and never go looking in any included modules. So our `PoliticalBook` class above is indeed correct, and we can happily ignore political clichés.

A similar "who wins?" question arises if we have the same method in two modules and include them both in the same class. To see this in action, imagine that we decide to create a slightly tongue-in-cheek writing-quality module especially for political writing:

```
module PoliticalWritingQuality

  # No phrase is too worn out to be a cliché
  # in political writing
```

```
  def number_of_cliches
    0
  end
end
```

Now what happens if we include both the original writing-quality module as well as the political version above in our `PoliticalBook` class?

```
class PoliticalBook < Document
  include WritingQuality
  include PoliticalWritingQuality

  # Lots of stuff omitted...

end
```

Which version of the `number_of_cliches` method will win? Again, the answer flows from the way modules get hooked into the class hierarchy of `PoliticalBook`: Include a module and it becomes the nearest parent "class" of the including class. Include a second module and *it* becomes the nearest parent of the including class, bumping the other module into second place. Have a look at Figure 16-2 to see this graphically. It's the methods in the most recently included module that always win. In our example, the `number_of_cliches` from `PoliticalWritingQuality` would be the one that `PoliticalBook` would end up with.

If you find yourself writing your own mixin module, there is one question that should be uppermost in your mind: What is the interface between my module and its including class? Since mixing in a module sets up an inheritance relationship between the including class and the module, you need to let your users know what that relationship is going to be before they start mixing. You should always add a few concise comments to your creation stating exactly what it expects from its including class:

```
# Methods to measure writing quality.
# Uses the content method of the including class.
module WritingQuality

  # Lots of stuff omitted...

end
```

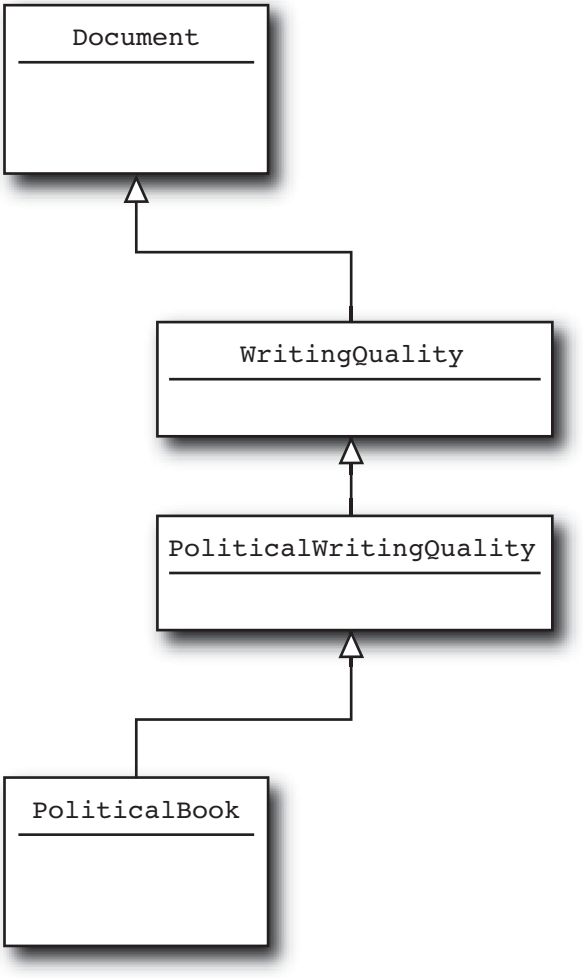

Figure 16-2 Modules get inserted sequentially into the inheritance tree.

A little guidance goes a long way to making your module useful.

In the Wild

Although the examples in this chapter have focused on modules containing a handful of methods, there really is no limit to what you can do with a module. DataMapper[3] is

3. DataMapper, which is maintained by a cast of, well, tens, can be found at www.datamapper.org.

a great example of just how far you can take a mixin module. DataMapper is an object relational mapper along the lines of ActiveRecord. Like ActiveRecord, DataMapper's purpose is to make objects persistable in a database. But unlike ActiveRecord, DataMapper does not require that your persistable objects extend any particular class. Instead, it's all done with a module. Here's what our `Document` class might look like as a DataMapper object:

```
class Document
  include DataMapper::Resource

  property :id, Integer, :serial => true
  property :title, String
  property :content, String
  property :author, String
end
```

By including one module—`DataMapper::Resource`—your class gains all the equipment it needs to persist itself in a database, and does so without using up the single superclass that Ruby allows for each class.

Rails also makes heavy use of mixin modules, most notably in the form of **helpers**. The Rails helper methods do exactly what their name suggests: They help you. For example, Rails includes a large number of helper methods that ease the pain of creating HTML, methods like `label` and `radio_button`, all delivered to your classes courtesy of mixin modules:

```
module ActionView

  # Huge amounts of code and helpful documentation omitted...

  module Helpers
    module FormHelper
      def label(object_name, method, text = nil, options = {})
         # ...
      end

      def radio_button(object_name, method, tag_value,options={})
        # ...
      end
    end
```

```
  end
end
```

Nor are methods all you can get from a mixin module. Mixin modules are also very convenient places to stash constants. Since including a module in a class inserts the module into the class hierarchy, the including class not only gains access to the module's methods but also to its constants. You can see this at work in the sqlite3 gem, which defines a series of modules full of nothing but constants. For example, there is a module that contains the list of error codes that might come back from sqlite3:[4]

```
module ErrorCode
  OK         =  0   # Successful result
  ERROR      =  1   # SQL error or missing database
  INTERNAL   =  2   # An internal logic error in SQLite
  PERM       =  3   # Access permission denied
  ABORT      =  4   # Callback routine requested an abort
  BUSY       =  5   # The database file is locked
  LOCKED     =  6   # A table in the database is locked

  # Seemingly endless list of remaining error codes omitted...
end
```

Now, as we saw in the last chapter, you can access all of these constants using their fully qualified names by saying things like `ErrorCode::OK` or `ErrorCode::Busy`. This can get tedious if you are doing it a lot; if so, you can drop the qualifying module name by simply including the `ErrorCode` module:

```
class SomeSQLiteApplication
  include ErrorCode

  def print_status_message( status )
    if status == ERROR
      puts "It failed!"
    elsif status == OK
      puts "It worked!"
```

4. In real life, the sqlite3 `ErrorCode` module is buried inside several other modules, which I have omitted here for the sake of brevity.

```
      else
        puts "Status was #{status}"
      end
    end
  end
```

Nothing like being able to insert modules into your class hierarchy to make the workday go faster!

Wrapping Up

In this chapter we have seen how mixin modules solve the problem of code that needs to be shared among classes without using up the one alloted superclass. We have also seen how mixin modules work, how they get drafted into service as a kind of phantom superclass when they are included into a host class. We have also seen that the key issue with mixins is the interface between the host class and the mixin module—what methods does the module provide and what does it expect the class to supply?

For many new Ruby programmers, mixin modules can act as a sort of skeleton key to the whole philosophy behind the language, the elegant way that Ruby tries to help programmers get code to execute where it is needed. Because of this, mixin modules pop up somewhere in most real-world Ruby applications. Nor are we done with them. Mixins will make an encore appearance in Chapter 20 when we look at a slick way to add both instance *and* class methods to a class with a single `include`. For now, however, we will turn to the code block, another way of getting code to where you need it.