Extracted from:

Metaprogramming Ruby 2

Program Like the Ruby Pros

The Pragmatic Bookshelf

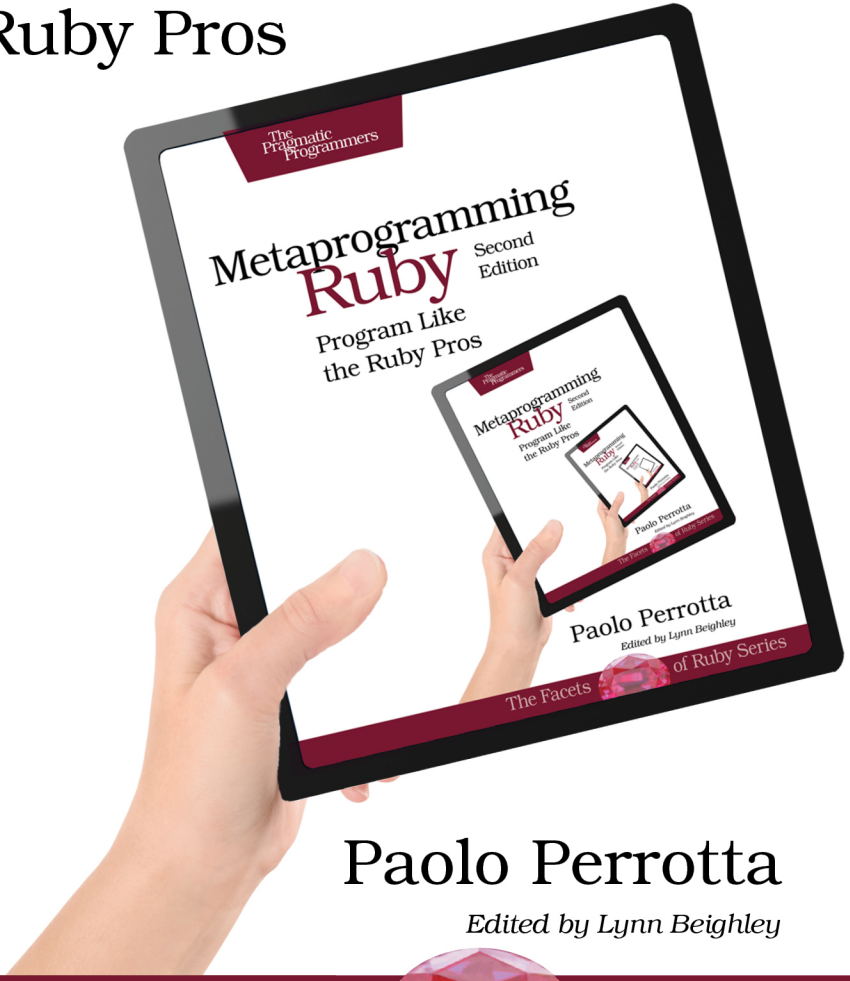
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Metaprogramming Ruby

Second Edition

Program Like the Ruby Pros



Paolo Perrotta

Edited by Lynn Beighley

The Facets of Ruby Series

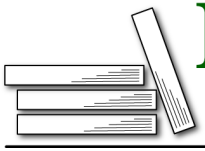
Metaprogramming Ruby 2

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 10

Active Support's Concern Module

In the previous chapter, you saw that the modules in Rails are special: when you include them, you gain both instance and class methods. How does that happen?

The answer comes from yet another module: Concern, in the Active Support library. ActiveSupport::Concern twists and bends the Ruby object model. It encapsulates the "add class methods to your includer" functionality, and it makes it easy to roll that functionality into other modules.

ActiveSupport::Concern is easier to understand if you know how it came to exist in the first place. We'll start by looking back at Rails' older versions, before Concern entered the scene.

Rails Before Concern

The Rails source code has changed a lot through the years, but some basic ideas haven't changed much. One of these is the concept behind ActiveRecord::Base. As you've seen in *ActiveRecord::Base*, this class is an assembly of dozens of modules that define both instance methods and class methods. For example, Base includes ActiveRecord::Validations, and in the process it gets instance and class methods.

The mechanism that rolls those methods into Base, however, has changed. Let's see how it worked in the beginning.

The Include-and-Extend Trick

Around the times of Rails 2, all validation methods were defined in ActiveRecord::Validations. (Back then, there was no Active Model library.) However, Validations pulled a peculiar trick:

gems/activerecord-2.3.2/lib/active_record/validations.rb

```
module ActiveRecord
  module Validations
    # ...

    def self.included(base)
      base.extend ClassMethods
      # ...
    end

    module ClassMethods
      def validates_length_of(*attrs) # ...
      # ...
    end

    def valid?
      # ...
    end

    # ...
  end
end
```

Does the code above look familiar? You've already seen this technique in *The VCR Example*, on page ?. Here's a quick recap. When ActiveRecord::Base includes Validations, three things happen:

1. The instance methods of Validations, such as valid?, become instance methods of Base. This is just regular module inclusion.

2. Ruby calls the included *Hook Method (?)* on Validations, passing ActiveRecord::Base as an argument. (The argument of included is also called base, but that name has nothing to do with the Base class—instead, it comes from the fact that a module's includer is sometimes called "the base class.")

3. The hook extends Base with the ActiveRecord::Validations::ClassMethods module. This is a *Class Extensions (?)*, so the methods in ClassMethods become class methods on Base.

As a result, Base gets both instance methods like valid? and class methods like validates_length_of.

This idiom is so specific that I hesitate to call it a spell. I'll refer to it as the *include-and-extend* trick. VCR borrowed it from Rails, as did many other Ruby projects throughout the years. Include-and-extend gives you a powerful way to structure a library: each module contains a well-isolated piece of functionality that you can roll into your classes with a simple include. That functionality can be implemented with instance methods, class methods, or both.

As clever as it is, include-and-extend has its own share of problems. For one, each and every module that defines class methods must also define a similar included hook that extends its includer. In a large codebase such as Rails', that hook was replicated over dozens of modules. As a result, people often questioned whether include-and-extend was worth the effort. After all, they observed, you can get the same result by adding one line of code to the includer:

```
class Base
  include Validations
  extend Validations::ClassMethods
  # ...
```

Include-and-extend allows you to skip the extend line and just write the include line. You might argue that removing this line from Base isn't worth the additional complexity in Validations.

However, complexity is not include-and-extend's only shortcoming. The trick also has a deeper issue—one that deserves a close look.

The Problem of Chained Inclusions

Imagine that you include a module that includes another module. You've seen an example of this in *The Validations Modules*: ActiveRecord::Base includes ActiveRecord::Validations, which includes ActiveModel::Validations. What would happen if both modules used the include-and-extend trick? You can find an answer by looking at this minimal example:

part2/chained_inclusions_broken.rb

```
module SecondLevelModule
  def self.included(base)
    base.extend ClassMethods
  end

  def second_level_instance_method; 'ok'; end

  module ClassMethods
    def second_level_class_method; 'ok'; end
  end
end

module FirstLevelModule
  def self.included(base)
    base.extend ClassMethods
  end

  def first_level_instance_method; 'ok'; end
```

```
  module ClassMethods
    def first_level_class_method; 'ok'; end
  end

  include SecondLevelModule
end

class BaseClass
  include FirstLevelModule
end
```

BaseClass includes FirstLevelModule, which in turn includes SecondLevelModule. Both modules get in BaseClass's chain of ancestors, so you can call both modules' instance methods on an instance of BaseClass:

```
BaseClass.new.first_level_instance_method      # => "ok"
BaseClass.new.second_level_instance_method     # => "ok"
```

Thanks to include-and-extend, methods in FirstLevelModule::ClassMethods also become class methods on BaseClass:

```
BaseClass.first_level_class_method             # => "ok"
```

SecondLevelModule also uses include-and-extend, so you might expect methods in SecondLevelModule::ClassMethods to become class methods on BaseClass. However, the trick doesn't work in this case:

```
BaseClass.second_level_class_method            # => NoMethodError
```

Go through the code step by step, and you'll see where the problem is. When Ruby calls SecondLevelModule.included, the base parameter is not BaseClass, but FirstLevelModule. As a result, the methods in SecondLevelModule::ClassMethods become class methods on FirstLevelModule—which is not what we wanted.

Rails 2 did include a fix to this problem, but the fix wasn't pretty: instead of using include-and-extend in both the FirstLevelModule and the SecondLevelModule, Rails used it only in the FirstLevelModule. Then FirstLevelModule#included forced the includer to also include the SecondLevelModule, like this:

part2/chained_inclusions_fixed.rb

```
module FirstLevelModule
  def self.included(base)
    base.extend ClassMethods
    base.send :include, SecondLevelModule
  end

  # ...
```

Distressingly, the code above made the entire system less flexible; it forced Rails to distinguish first-level modules from other modules, and each module

had to know whether it was supposed to be first-level. (To make things clumsier, Rails couldn't call Module#include directly, because it was a private method—so it had to use a *Dynamic Dispatch (?)* instead. Recent rubies made include public, but we're talking ancient history here.)

At this point in our story, you'd be forgiven for thinking that include-and-extend created more problems than it solved in the first place. This trick forced multiple modules to contain the same boilerplate code, and it failed if you had more than one level of module inclusions. To address these issues, the authors of Rails crafted ActiveSupport::Concern.

ActiveSupport::Concern

ActiveSupport::Concern encapsulates the include-and-extend trick and fixes the problem of chained inclusions. A module can get this functionality by extending Concern and defining its own ClassMethods module:

part2/using_concern.rb

```
require 'active_support'

module MyConcern
  extend ActiveSupport::Concern

  def an_instance_method; "an instance method"; end

  module ClassMethods
    def a_class_method; "a class method"; end
  end
end

class MyClass
  include MyConcern
end

MyClass.new.an_instance_method  # => "an instance method"
MyClass.a_class_method          # => "a class method"
```

In the rest of this chapter I'll use the word "concern" with a lowercase C to mean "a module that extends ActiveSupport::Concern," like MyConcern does in the example above. In modern Rails, most modules are concerns, including ActiveRecord::Validations and ActiveModel::Validations.

Let's see how Concern works its magic.

A Look at Concern's Source Code

The source code of Concern is quite short but also fairly complicated. It defines just two important methods: extended and append_features. Here is extended:

gems/activesupport-4.1.0/lib/active_support/concern.rb
```
module ActiveSupport
  module Concern
    class MultipleIncludedBlocks < StandardError #:nodoc:
      def initialize
        super "Cannot define multiple 'included' blocks for a Concern"
      end
    end

    def self.extended(base)
      base.instance_variable_set(:@_dependencies, [])
    end

    # ...
```

When a module extends Concern, Ruby calls the extended *Hook Method (?)*, and extended defines an @_dependencies *Class Instance Variable (?)* on the includer. I'll show you what happens to this variable in a few pages. For now, just remember that all concerns have it, and it's initially an empty array.

To introduce Concern#append_features, the other important method in Concern, let me take you on a very short side-trip into Ruby's standard libraries.

Module#append_features

Module#append_features is a core Ruby method. It's similar to Module#included, in that Ruby will call it whenever you include a module. However, there is an important difference between append_features and included: included is a Hook Method that is normally empty, and it exists only in case you want to override it. By contrast, append_features is where the real inclusion happens. append_features checks whether the included module is already in the includer's chain of ancestors, and if it's not, it adds the module to the chain.

There is a reason why you didn't read about append_features in the first part of this book: in your normal coding, you're supposed to override included, not append_features. If you override append_features, you can get some surprising results, as in the following example:

part2/append_features.rb
```
module M
  def self.append_features(base); end
end

class C
  include M
end

C.ancestors   # => [C, Object, Kernel, BasicObject]
```

As the code above shows, by overriding append_features you can prevent a module from being included at all. Interestingly, that's exactly what Concern wants to do, as we'll see soon.

Concern#append_features

Concern defines its own version of append_features.

gems/activesupport-4.1.0/lib/active_support/concern.rb

```
module ActiveSupport
  module Concern
    def append_features(base)
        # ...
```

Remember the *Class Extension (?)* spell? append_features is an instance method on Concern, so it becomes a class method on modules that extend Concern. For example, if a module named Validations extends Concern, then it gains a Validation.append_features class method. If this sounds confusing, look at this picture showing the relationships between Module, Concern, Validations, and Validation's singleton class:

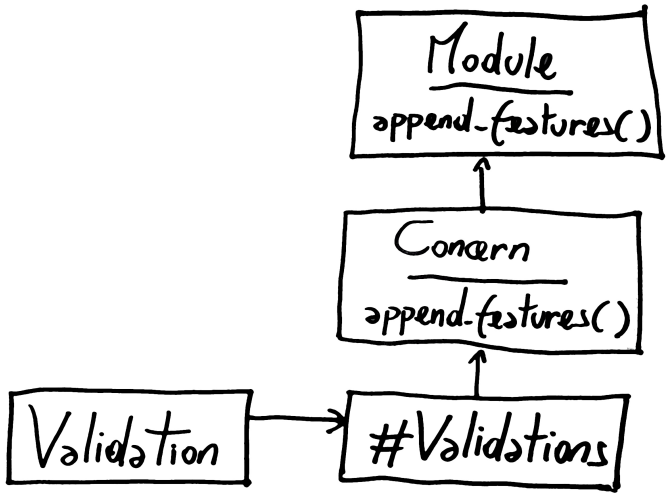


Figure 10—ActiveSupport::Concern overrides Module#append_features.

Let's recap what we've learned so far. First, modules that extend Concern get an @_dependencies Class Variable. Second, they get an override of append_features. With those two concepts in place, we can look at the code that makes Concern tick.

Inside Concern#append_features

Here is the code in Concern#append_features:

gems/activesupport-4.1.0/lib/active_support/concern.rb

```
module ActiveSupport
  module Concern
    def append_features(base)
      if base.instance_variable_defined?(:@_dependencies)
        base.instance_variable_get(:@_dependencies) << self
        return false
      else
        return false if base < self
        @_dependencies.each { |dep| base.send(:include, dep) }
        super
        base.extend const_get(:ClassMethods) \
          if const_defined?(:ClassMethods)
        # ...
      end
    end

    # ...
```

This is a hard piece of code to wrap your brain around, but its basic idea is simple: never include a concern in another concern. Instead, when concerns try to include each other, just link them in a graph of dependencies. When a concern is finally included by a module that is not itself a concern, roll all of its dependencies into the includer in one fell swoop.

Let's look at the code step by step. To understand it, remember that it is executed as a class method of the concern. In this scope, self is the concern, and base is the module that is including it, which might or might not be a concern itself.

When you enter append_features, you want to check whether your includer is itself a concern. If it has an @_dependencies Class Variable, then you know it is a concern. In this case, instead of adding yourself to your includer's chain of ancestors, you just add yourself to its list of dependencies, and you return false to signal that no inclusion actually happened. For example, this happens if you are ActiveModel::Validations, and you get included by ActiveRecord::Validations.

What happens if your includer is *not* itself a concern—for example, when you are ActiveRecord::Validations, and you get included by ActiveRecord::Base? In this case, you check whether you're already an ancestor of this includer, maybe because you were included via another chain of concerns. (That's the meaning of base < self.) If you are not, you come to the crucial point of the entire exercise: you recursively include your dependencies in your includer. This minimalistic

dependency management system solves the issue that you've read about in *The Problem of Chained Inclusions*, on page 9.

After rolling all your dependent concerns into your includer's chain of ancestors, you still have a couple of things to do. First, you must add *yourself* to that chain of ancestors, by calling the standard `Module.append_features` with `super`. Finally, don't forget what this entire machinery is for: you have to extend the includer with your own `ClassMethods` module, like the include-and-extend trick does. You need `Kernel#const_get` to get a reference to `ClassMethods`, because you must read the constant from the scope of `self`, not the scope of the `Concern` module, where this code is physically located.

`Concern` also contains some more functionality, but you've seen enough to grasp the idea behind this module.