CHAPTER 22

Use method_missing for Delegation

About ten years ago, for reasons that are still inexplicable to me, I gave up software development. I put down my keyboard and donned the coat if not the actual tie of a software development manager. I prepared schedules. I ran meetings. I did performance reviews. And I was miserable. The more time I spent with people, the more I loved my computer. I look back on those two or three years as the most difficult period of my professional life. Instead of rolling into work every morning, eager to do battle with the essential complexity of the universe, I dragged myself to the office to fight the pervasive stupidity of a Byzantine organization.

Still, my years in organizational purgatory were not completely wasted. During my stint as a manager I did learn some things, mainly things about people. The most important thing that I learned was that you just have to trust the folks who work for you. Make sure they know what they're doing. Make sure they know what you want them to do. Then get out of the way and let them do it. In short, I learned to delegate.

Delegation is important in object oriented programming too. In the programming world delegation is the idea that an object might secretly use another object to get part of the job done. Since getting out of the way is as important in the world of programs as it is in the real word, in this chapter we will look at doing delegation via `method_missing`. We will see that `method_missing` provides an almost painless mechanism for delegating calls from one object to another. We will also look at some of the dangers of delegating with `method_missing` and see whether we can find out how to balance those dangers with all the power that `method_missing` gives us.

The Promise and Pain of Delegation

Delegation—the coding edition—is a pretty basic concept: Sometimes you find yourself building an object that wants to do something and you happen to have another object that does exactly that something. You *could* copy all of the code from one class to the other, but that is probably a bad idea.[1] Instead, what you do is delegate: You supply the first object with a reference to the second, and every time you need to do that something you call the right method on the other object. Delegation is just another word for foisting the work on another object.

To make this more real, imagine that some secret spy agency has started using our `Document` class to store sensitive material. In fact, the material is so sensitive that *The Agency* would like to be able to create a special read-only version of any document, but a read-only version with a twist: Any program that gains access to one of these special documents is only allowed to see the document for five seconds. Any longer and the document should become unavailable. Oh, and the documents are liable to change anytime, so you can't just copy the original document. What's a coder to do?

Clearly, some sort of document wrapper is called for:

```
class SuperSecretDocument
  def initialize(original_document, time_limit_seconds)
    @original_document = original_document
    @time_limit_seconds = time_limit_seconds
    @create_time = Time.now
  end

  def time_expired?
    Time.now - @create_time >= @time_limit_seconds
  end

  def check_for_expiration
    raise 'Document no longer available' if time_expired?
  end

  def content
    check_for_expiration
    return @original_document.content
  end
```

1. "Bad" in this context means abysmally horrible.

```
    def title
      check_for_expiration
      return @original_document.title
    end

    def author
      check_for_expiration
      return @original_document.author
    end

    # and so on...
  end
```

The SuperSecretDocument class holds onto a reference to the original Document instance and a time limit. As long as the time has not expired, the SuperSecretDocument will delegate any method calls off to the original document. Once the time is up, SuperSecretDocument stops cooperating and will only return an exception. Armed with the code above, we can now create some satisfyingly perishable documents:

```
  original_instructions = get_instructions
  instructions = SuperSecretDocument.new(original_instructions, 5)
```

Execute the preceding code and your instructions will self destruct in five seconds.[2]

The Trouble with Old-Fashioned Delegation

The trouble with this traditional style of delegation is that it is the programming equivalent of the manager who gives someone a job to do and then insists on supervising every detail.

To see the problem, imagine that our Document class was less of a toy and supported more of the features that you would find on a real document, features like page layout (landscape or portrait?), size (A4 or U.S. letter?), and so on. The trouble is that our SuperSecretDocument class needs to grow right along with the regular Document class:

2. With apologies to *Mission Impossible*—the old TV series, you understand, not those dreadful movies.

```
class SuperSecretDocument
  def initialize(original_document, time_limit_seconds)
    @original_document = original_document
    @time_limit_seconds = time_limit_seconds
    @create_time = Time.now
  end

  def time_expired?
    Time.now - @create_time >= @time_limit_seconds
  end

  def check_for_expiration
    raise 'Document no longer available' if time_expired?
  end

  # content, title and author methods omitted
  # to keep from kill even more trees...

  # And some new methods....

  def page_layout
    check_for_expiration
    return @original_document.page_layout
  end

  def page_size
    check_for_expiration
    return @original_document.page_size
  end

  # And so on and so on and so on...
end
```

The problem with this lengthy stretch of delegating code is that your program isn't really getting all of the benefits of delegation. Yes, it's the `Document` instance that's really doing the work of getting the revision dates and paper sizes, but the `SuperSecretDocument` object is always there, looking over the document's shoulder with all of that dull, delegating code. In programming as in management, the key to delegation is getting out of the way.

The method_missing Method to the Rescue

The secret to getting out of the way lies in method_missing. Think about what would happen if we took all of those repetitious delegating methods out of the SuperSecretDocument class. Without the delegating methods, every time someone called a Document method on a SuperSecretDocument instance, they would be calling a method that wasn't there. Since the method is missing, Ruby would eventually call method_missing. Herein lies an opportunity: Instead of simply logging a message or raising an exception in method_missing, we can use a call to method_missing as an chance to delegate to the real Document:

```
class SuperSecretDocument
  def initialize(original_document, time_limit_seconds)
    @original_document = original_document
    @time_limit_seconds = time_limit_seconds
    @create_time = Time.now
  end

  def time_expired?
    Time.now - @create_time >= @time_limit_seconds
  end

  def check_for_expiration
    raise 'Document no longer available' if time_expired?
  end

  def method_missing(name, *args)
    check_for_expiration
    @original_document.send(name, *args)
  end
end
```

This new, and much briefer, version of SuperSecretDocument uses method_missing to catch all of the calls that need to be delegated to the original document. When the SuperSecretDocument method_missing catches a method call it uses the send method to forward the call onto the original document:

```
    @original_document.send(name, *args)
```

Recall that we saw the send method back in Chapter 7, where we used it to get around the restrictions of private and protected methods. In the code above we are using send in its full glory as sort of the inverse of method_missing: While method_missing lets you catch arbitrary method calls from inside of a class, send lets you make arbitrary method calls on some other object. Best of all, the arguments for send, the name of the method (as a symbol) followed by the arguments to the method, line up exactly with the arguments to method_missing.

One obvious question with using this technique is, what happens when (inevitably) there is a real screwup, when someone accidentally calls instructions.continent instead of instruction.content? A little reflection will show that this is not really a problem. Since there is no continent method on the SuperSecretDocument instance, the call will get forwarded to the real Document instance. And since there is no continent there either, it will obligingly raise an exception.

The huge advantage of our new SuperSecretDocument implementation is that it is small—the whole class is under 20 lines—and doesn't need to grow as we add new methods to the Document class. The method_missing method will catch whatever methods you throw at SuperSecretDocument and will forward them, whatever they are, to the Document instance.

In fact, SuperSecretDocument is so generic that it isn't really Document specific at all. We could, for example, use SuperSecretDocument to wrap a String:

```
string = 'Good morning, Mr. Phelps'
secret_string = SuperSecretDocument.new( string, 5 )

puts secret_string.length          # Works fine
sleep 6
puts secret_string.length          # Raises an exception
```

The SuperSecretDocument class is effectively a perishable container for any object you might come up with.

More Discriminating Delegation

Although SuperSecretDocument will indiscriminately forward any method that comes its way, there is nothing to prevent us from doing a more selective job of delegation. We might, for instance, decide that we want SuperSecretDocument to deal only with a narrowly defined set of methods:

```
class SuperSecretDocument
  # Lots of code omitted...

  DELEGATED_METHODS = [ :content, :words ]

  def method_missing(name, *args)
    check_for_expiration
    if DELEGATED_METHODS.include?( name )
      @original_document.send(name, *args)
    else
      super
    end
  end
end
```

This rendition of SuperSecretDocument has a list of the names of the methods it wants to delegate to @original_document. If a method call comes in for some other method, we just call super, which forwards the original method call (arguments and all!) up the class hierarchy where it will eventually meet its fate with a NameError exception.

Staying Out of Trouble

There is one nasty blemish on the otherwise smooth finish that is method_missing-based delegation: What if the method is not actually missing? To see what I mean, ask yourself what would happen if we had an instance of the original SuperSecretDocument class—the one that just delegates everything—and we called to_s on it:

```
original_instructions = get_instructions
instructions = SuperSecretDocument.new(original_instructions, 5)
puts instructions.to_s
```

You might expect that this code would do one of two things: either call the to_s method on the Document instance or, if the time has expired, simply blow up. What actually happens instead is that you end up calling the SuperSecretDocument version of the to_s method, so that the output would be something like:

```
#<SuperSecretDocument:0x87273ac>
```

The trouble is that there actually *is* a to_s method on instances of SuperSecret-Document: They inherit it from the Object class. The same goes for all of the other methods your delegating object might have. If a delegating object actually has a method, the way our SuperSecretDocument instances all have a to_s method, then the method is not actually missing and method_missing is not going to go off for that method.

There is an easy way out of this conundrum, BasicObject. Recall from Chapter 7 that BasicObject was introduced in Ruby 1.9 and is the superclass of Object. As the name suggests, BasicObject is very stripped down: Instances of BasicObject inherit only a handful of methods. This means that BasicObject is an ideal candidate to start with when you are doing the kind of mass delegation we are looking for with SuperSecretDocument. Thus, if we redefined SuperSecretDocument to be a subclass of BasicObject:

```
class SuperSecretDocument < BasicObject
  # Most of the class omitted...
  def initialize(original_document, time_limit_seconds)
    @original_document = original_document
    @time_limit_seconds = time_limit_seconds
    @create_time = ::Time.now
  end

  def time_expired?
    ::Time.now - @create_time >= @time_limit_seconds
  end

  def check_for_expiration
    raise 'Document no longer available' if time_expired?
  end

  def method_missing(name, *args)
    check_for_expiration
```

```
      @original_document.send(name, *args)
    end
  end
```

Then the to_s method will time out just like title and content.[3]

In the Wild

The Ruby standard library comes with a very handy method_missing-based delegation utility in the delegate.rb file. This file contains a number of classes that take what little sting there is in delegating with method_missing. The simplest one of the bunch is probably the aptly named SimpleDelegator that you can use as a superclass for your delegating class. All you need to do is call the SimpleDelegator constructor with the object you are delegating to, and it will take care of the rest. Here, for example, is a SimpleDelegator based do-nothing wrapper for our Document class:

```
  require 'delegate'

  class DocumentWrapper < SimpleDelegator
    def initialize( real_doc )
      super( real_doc )
    end
  end
```

That's pretty much it. With just seven lines of code, we have a fully functional wrapper for any document:

```
  text =  'The Hare was once boasting of his speed...'
  real_doc = Document.new( 'Hare & Tortoise', 'Aesop', text )

  wrapper_doc = DocumentWrapper.new( real_doc )
```

3. Sadly, it seems that in software engineering there is always at least a little catch. If you look carefully at the BasicObject version of SuperSecretDocument, you will see that we needed to explicitly specify the scope of the Time class with ::. We need to do this because of the disconnected role of BasicObject as the sort of noble gas of Ruby classes.

Then any call to `wrapper_doc` will behave just like a call to the `real_doc`, so that running this:

```
puts wrapper_doc.title
puts wrapper_doc.author
puts wrapper_doc.content
```

Will print:

```
Hare & Tortoise
Aesop
The Hare was once boasting of his speed...
```

Aside from `delegate.rb`, the quintessential example of delegation by `method_missing` is probably the one you will find in ActiveRecord. Early versions of ActiveRecord used `method_missing`-based delegation to return the values of fields from a row in a table. For example, if you used ActiveRecord to find a row in a table, something like this:

```
the_employee = Employee.find( :first )
```

Then hidden inside of that record would be a hash containing the field values from the database, perhaps `{ :first_name => 'Bob', :last_name => 'Kiel' }`. You could then get at the fields as though they were ordinary methods:

```
puts the_employee.first_name
puts the_employee.last_name
```

This is slick enough, but more recent versions of ActiveRecord do something slicker still. In late-model ActiveRecord versions, the first time you access `the_employee.first_name`, the `method_missing` method will go off just like it did in the olden days. But instead of simply looking up the field value, the newer `method_missing` will also define the `first_name` and (for good measure) `last_name` methods on the class. It's these newly defined methods that get used on subsequent calls. Apparently, skipping the `method_missing` rigmarole improves performance enough to make the whole thing worthwhile. It is also impressive to watch.

Finally, if you happen to still be using a 1.8.X version of Ruby and need something like `BasicObject` for your delegating needs, don't despair. The blankslate gem provides, via the magic of Ruby metaprogramming, a more than adequate simulation of `BasicObject`.

Wrapping Up

In this chapter we explored the wonders of delegation via `method_missing`. We saw how you can put `method_missing` to work for something other than error handling. We also saw how easy it is to use `method_missing` to build a very painless delegation mechanism. Far from being a simple error-handling facility, `method_missing` provides the Ruby programmer with a very general-purpose way of catching and interpreting method calls. In the next chapter we will build on this idea to see how we can put `method_missing` to an even more exotic use: answering calls to methods that you may have never dreamed of.