Chapter 10

Construct Your Classes from Short, Focused Methods

Let's face it: Despite shelves full of books on software architecture, enough UML diagrams to fill an art museum, and design meetings that seem to last longer than the pyramids, building software mostly comes down to writing one method after another.

An important, but frequently ignored question is, how exactly should we break our classes up into those methods? Although the question of "How should I write my methods?" is not really one that a programming language can answer, programming languages do tend to encourage one style over another. In this chapter we are going to look at how Ruby programmers tend to write methods. We will see that most Ruby programmers favor very short methods, methods that stick to doing one thing and doing it well. We will see that breaking your code up into many short, single-purpose methods not only plays to the strengths of the Ruby programming language but also makes your whole application more testable.

Compressing Specifications

Imagine that you just landed a job working for the Universal Software Specification Repository. You and your colleagues are tasked with archiving every software spec that has ever been written. Since the industry has spewed out a lot of specs over the years, the plan is to compress the documents before they are stored. Your job is to implement

the heart of the compression algorithm, which will take in a text string and produce two arrays, which will then be stored in the archive. The first array will contain all the unique words in the original text. Thus, if you start with this text:

```
This specification is the specification for a specification
```

Then the first array would contain all the words found in the text, with no repeats, like this:

```
["This", "specification", "is", "the", "for", "a"]
```

The second array will contain integer indexes. There will be one index in this second array for each word in the original document. In our example, the second array would contain this:

```
[0, 1, 2, 3, 1, 4, 5, 1]
```

We can use these two data structures to reproduce the original sequence of words by running through the index array, looking up each word in the unique words array as we go. Thus we can tell that the fourth word of our text is `"the"`, since the fourth index is 3 and `"the"` is the word at index 3 in the unique words array. This technique is the sort of thing you might do when you want to compress some text that contains a lot of long, repeated words—which is more or less the definition of a software specification.

Doing a passable job of taking some text apart and compressing it into the two arrays is not difficult. Here's our first stab:

```
class TextCompressor
  attr_reader :unique, :index

  def initialize( text )
    @unique = []
    @index = []

    words = text.split
    words.each do |word|
      i = @unique.index( word )
      if i
        @index << i
```

```
        else
          @unique << word
          @index << unique.size - 1
        end
      end
    end
  end
```

Using the `TextCompressor` class is simplicity itself: You just supply some text to the constructor:

```
text = "This specification is the spec for a specification"
compressor = TextCompressor.new( text )
```

And then you can get at the arrays of unique words and the word indexes, presumably for storage elsewhere:

```
unique_word_array = compressor.unique
word_index = compressor.index
```

From many points of view, the code here is just fine: It does work,[1] and it is fairly simple. The trouble with this first attempt is that the code itself doesn't do a very good job of speaking to the humans who need to maintain it. The next programmer who tries to understand `TextCompressor` will need to spend some time looking hard at the `initialize` method. It would be nice if the code, along with working, would also do a better job of explaining itself to the next engineer.

Let's see if we can't make the code a little more articulate:

```
class TextCompressor
  attr_reader :unique, :index

  def initialize( text )
    @unique = []
    @index = []
```

1. Well, it does lose all the white space in the text and won't react well to punctuation. Close enough for an example.

```
    words = text.split
    words.each do |word|
      i = unique_index_of( word )
      if i
        @index << i
      else
        @index << add_unique_word( word )
      end
    end

  def unique_index_of( word )
    @unique.index(word)
  end

  def add_unique_word( word )
    @unique << word
    unique.size - 1
  end
end
```

What we have done in this second pass is to pull out the code that looks up a word in the unique array into the `unique_index_of` method. We have also pulled out the code that adds a new word to the `@unique` array into the `add_unique_word` method.

This second version is definitely an improvement: At least the names of the new methods give us a clue as to what's going on in the `initialize` method. But we are not done yet: Our new `initialize` method is still fairly obtuse and, worse, it seems a bit unbalanced. On the one hand `initialize` is acting like a "don't bother me with the details" boss by delegating work to `unique_index_of` and `add_unique_word`, but then it turns around and gets involved in the dirty details of managing the `@index` array.

Let's take one more shot at it:

```
class TextCompressor
  attr_reader :unique, :index

  def initialize( text )
    @unique = []
    @index = []
```

```
    add_text( text )
  end

  def add_text( text )
    words = text.split
    words.each { |word| add_word( word ) }
  end

  def add_word( word )
    i = unique_index_of( word ) || add_unique_word( word )
    @index << i
  end

  def unique_index_of( word )
    @unique.index(word)
  end

  def add_unique_word( word )
    @unique << word
    unique.size - 1
  end
end
```

We've done a number of things with this latest version: The `initialize` method now foists the task of doing the compression onto the `add_text` method, which itself mostly delegates to `add_word`. In addition, by cutting down on the clutter, it became apparent that we could collapse the original "is the word there or not?" if statement into an or[2] expression, which ends up in the `add_word` method.

Composing Methods for Humans

What we have just done to our `TextCompressor` class is to apply the **composed method** technique to it. The composed method technique advocates dividing your class up into methods that have three characteristics. First, each method should do a single thing—focus on solving a single aspect of the problem. By concentrating on one thing, your methods are not only easier to write, they are also easier to understand.

2. That's or as in ||.

Second, each method needs to operate at a single conceptual level: Simply put, don't mix high-level logic with the nitty-gritty details. A method that implements the business logic around, say, currency conversions, should not suddenly veer off into the details of how the various accounts are stored in a database.

Finally, each method needs to have a name that reflects its purpose. Nothing new here; we have all heard endless lectures about picking good method names. The time to listen to all of that haranguing is when you are creating lots of little methods that you are trying to pull together into a functional whole. Done right, the method names guide you through the logic of the code.

Take another look at the final version of the compression code and you will see that it follows all three maxims. There's a method to add some text, one to add a single word, and one to find the index of a word that's already there. A little more subtly, the code in each of the new methods is at a single conceptual level: Some methods, `add_unique_word` for example, are down in the weeds dealing with the messy details of array indexes, while other methods, particularly `add_text`, operate at a higher conceptual level. Aside from being compact and focused, each of our methods also has a carefully chosen name, a name that tells the reader exactly what the method does.

Why is building small, well-named methods that do one thing such a good idea? It's not about writing better code for the computer, because the computer doesn't care. You can code the same algorithm in a handful of large methods or in a myriad of little methods and, as long as you've gotten the details right, the computer will give you exactly the same answer. The reason you should lean toward smaller methods is that all those compact, easy-to-comprehend methods will help *you* get the details right.

Composing Ruby Methods

People have been writing short, coherent methods in a whole range of programming languages for years. The reason we are talking about it here is that, like testing, the Ruby community has taken to composing their methods with a vengeance. Look at a class, and if it contains a lot of short, pointed methods it just feels like Ruby.

Nor is this simply a cultural phenomenon: The composed method way of building classes is particularly effective in Ruby because Ruby is such a "low ceremony" language. In Ruby, the cost of defining a new method is very low: just an additional `def` and an extra `end`. Since defining a new Ruby method adds very little noise to your code, in Ruby you can get the full composed method bang for a very modest code overhead buck.

Having many fine-grained methods also tends to make your classes easier to test. Consider that with the original version of `TextCompressor`, the only thing you could test was, well, everything. You fed some text into the class and that monolithic `initialize` method took over. Contrast that with our latest version that allows us to test all sorts of things:

```
describe TextCompressor do

  it "should be able to add some text" do
    c = TextCompressor.new( '' )
    c.add_text( 'first second' )
    c.unique.should == [ 'first', 'second' ]
    c.index.should == [ 0, 1 ]
  end

  it "should be able to add a word" do
    c = TextCompressor.new( '' )
    c.add_word( 'first' )
    c.unique.should == [ 'first' ]
    c.index.should == [ 0 ]
  end

  it "should be able to find the index of a word" do
    c = TextCompressor.new( 'hello world' )
    c.unique_index_of( 'hello' ).should == 0
    c.unique_index_of( 'world' ).should == 1
  end

  # ...
end
```

By decomposing your class into a lot of small methods, you provide a much larger number of sockets into which you can plug your tests.

One Way Out?

Short, easily comprehended methods also have some secondary advantages as well. Take the old bit of coding advice that every method should have exactly one way out, so that all of the logic converges at the bottom for a single return. Suppose, for example,

we needed a method to rate the text in a document, based on the number of pretentious or slangy words in the document. If we construct a single, longish method and sprinkle returns here and there, we end up with code that is definitely hard to follow:

```
class Document

  # Most of class omitted...

  def prose_rating
    if pretentious_density > 0.3
      if informal_density < 0.2
        return :really_pretentious
      else
        return :somewhat_pretentious
      end
    elsif pretentious_density < 0.1
      if informal_density > 0.3
        return :really_informal
      end
      return :somewhat_informal
    else
      return :about_right
    end
  end

  def pretentious_density
    # Somehow compute density of pretentious words
  end

  def informal_density
    # Somehow compute density of informal words
  end

end
```

The code above is not horrible, but I suspect you would have to stare at it awhile to really get a feeling for the flow. Now take a look at a single-return rewrite of the same method:

```
  def prose_rating
    rating = :about_right
```

```
  if pretentious_density > 0.3
    if informal_density < 0.2
      rating = :really_pretentious
    else
      rating = :somewhat_pretentious
    end
  elsif pretentious_density < 0.1
    if informal_density > 0.3
      rating = :really_informal
    else
      rating = :somewhat_informal
    end
  end

  rating
end
```

It's not that dramatic, but the single-exit version does seem more comprehensible—you don't find yourself scanning every line trying to figure out if the method is suddenly going to exit right there on you. So, problem solved, right?

Not really. The real problem is that the first version of the `prose_rating` method goes on and on for way too long. The second version is marginally better, but only just. A better fix is to attack the "hard to understand" problem head-on with some composed methods:

```
def prose_rating
  return :really_pretentious if really_pretentious?
  return :somewhat_pretentious if somewhat_pretentious?
  return :really_informal if really_informal?
  return :somewhat_informal if  somewhat_informal?
  return :about_right
end

def really_pretentious?
  pretentious_density > 0.3 && informal_density < 0.2
end

def somewhat_pretentious?
  pretentious_density > 0.3 && informal_density >= 0.2
end
```

```
def really_informal?
  pretentious_density < 0.1 && informal_density > 0.3
end

def somewhat_informal?
  pretentious_density < 0.1 &&  informal_density <= 0.3
end

def pretentious_density
  # Somehow compute density of pretentious words
end

def informal_density
  # Somehow compute density of informal words
end
```

Now that we have broken `prose_rating` into something more digestible, it really doesn't matter whether it has one return or whether—as in this last version—it is made up entirely of returns. Once you can take in the whole method in single glance, then the motivation for the single-return rule goes away.

Staying Out of Trouble

The key to preventing your composed methods from turning on you is to remember that every method should have two things going for it. First, it should be short. And second, it should be coherent. In plain English, your method should be compact but it should also *do something*. Unfortunately, since short is so much easier to remember than coherent, programmers will sometimes go too far in breaking up their methods. For example, we might set out to decompose the `add_unique_word` method even further and end up with this:

```
class TextCompressor
  # ...

  def add_unique_word( word )
    add_word_to_unique_array( word )
    last_index_of_unique_array
  end

  def add_word_to_unique_array( word )
```

```
      @unique << word
    end

    def last_index_of_unique_array
      unique.size - 1
    end
  end
```

The two new methods do add something to the class. Unfortunately, the word for this addition is *clutter*. Taken to the dysfunctional extreme, it's possible to compose method yourself into a diffuse cloud of programming dust. Don't do that.

In the Wild

Getting into the habit of writing the short, pointy methods technique takes time and effort. As you go through the process, it's easy to get frustrated and decide that it's just impossible to get anything done with such tiny chunks of logic. But like a lot of conclusions born of frustration, this one is just not correct. If you need a good role model to motivate your forays into composed method land, consider `ActiveRecord::Base`.

If you are a Rails programmer, then you know that it is `ActiveRecord::Base` that turns this:

```
class Employee < ActiveRecord::Base
end
```

Into a functionally rich interface to a database table. Yet a quick look at `base.rb`, which defines the core of the `ActiveRecord::Base` class, turns up about 1800 non-comment lines that define about 200 methods. Do the math and you have an average of about nine lines of code per method. Here, for example is the key `find` method, all 11 non-blank lines of it:

```
  def find(*args)
    options = args.extract_options!
    validate_find_options(options)
    set_readonly_option!(options)

    case args.first
    when :first then find_initial(options)
```

```
    when :last  then find_last(options)
    when :all   then find_every(options)
    else             find_from_ids(args, options)
    end
  end
```

Not only are the `ActiveRecord::Base` methods generally short, but they also follow the other composed method recommendation: They sport very descriptive names. Care to venture a guess as to what the `find_last` or `find_every` methods in the code above do? `ActiveRecord::Base` is the existence proof for the postulate that you can get something done—in this case, quite a lot done—with short, focused methods.

Wrapping Up

In this chapter we have looked at the composed method technique, which advocates building short, focused methods. While the composed method technique is not technically part of the Ruby programming language, it is a key tool that Ruby programmers use to construct code. Building your methods this way really comes down to writing short, focused methods, each with a name that tells the reader exactly what it does. Not only will these short methods be easier to understand, they will also be easier to test.