# CHAPTER 27
# Invent Internal DSLs

Rake. RSpec. ActiveRecord models. A build tool, a testing utility, and a database interface library. It seems unlikely that these three chunks of code could have anything in common, but they do. All three are examples of a particular style of programming, the internal Domain Specific Language, or DSL. More than any other technique, the internal DSL has come to represent the easy eloquence of Ruby; each is an illustration of how the language allows you to create tools that can solve whole classes of problems.

In this chapter we will look at building DSLs in Ruby. Actually, build is probably the wrong word: What we will really see is how you can *grow* a DSL, how you can start with an ordinary API and slowly transform it into something more, so that step by step it becomes its own little language. Along the way we will see how to pull together a number of the Ruby programming techniques covered in previous chapters into a really powerful combination. Finally, we will look at when an internal DSL is a good solution and when it is better to turn to something else.

## Little Languages for Big Problems

Software engineering is all about trade-offs. There rarely is a "best" or "correct" solution to programming problems. We keep all the data in memory and the program runs faster—at the cost of all that memory. Or, we strike a different bargain and save the memory at the price of speed. We build plain-looking GUIs that work now or we create fancy ones that take a little longer. We drink the free but dubious coffee in the office or we go out and spend a few dollars for the real thing.

You see this kind of trade-off in the design of programming languages. The typical general-purpose programming language is good at solving a huge range of problems. C# is about as good at writing accounting systems as it is at building software that will predict earthquakes. You can use Java to build huge enterprise applications or tiny cell phone GUIs. Unfortunately, there is a price to be paid for being general purpose. A language that tries to do everything can't afford to be great at any one thing.

**Domain specific languages**, or **DSLs** for short, strike a different sort of bargain. Instead of being pretty good at a lot of things, a DSL tries to be really great at one narrowly defined class of problems. Imagine creating a programming language that pulls out all of the stops to make it easy to build earthquake prediction systems. But specialization has its own cost. The price is that a language that's great at earthquakes is probably going to be lousy at doing accounting. Still, if your problem is earthquakes, then a bit of inflexibility may be worth it.

If you do decide to go the DSL route, you have a choice to make. The traditional way to build a DSL is to get out your copy of "Parsers and Compilers for Beginners" and start coding a whole new language. Martin Fowler calls this traditional approach the **external DSL**, external in the sense that the new language is separate or external from the implementation language. The downside of the external DSL approach is right out in the open: You need to build a whole new programming language. Will you have `if` statements? Classes and methods? What will you use for a comment character? Building a brand-new programming language from scratch is not something to be undertaken lightly.

The alternative is to build your DSL atop an existing language. You add so much support for solving problems in your chosen domain into an existing language that it starts to feel like a specialized tool. The beauty of this second approach is that you don't need to recreate all of the plumbing of a programming language—it's already there for you. Fowler's term for this other kind of DSL is, logically enough, the **internal DSL**. Internal DSLs are internal in that the DSL is built right into the implementation language. The good news is that Ruby, with its "the programmer is always right" feature set and very flexible syntax, makes a great platform for building internal DSLs.

## Dealing with XML

To turn all this theory into real code, let's put down our familiar and somewhat contrived `Document` class and look at a very real-world problem: XML. These days, XML is so common that it is a rare software engineer who manages to avoid it.[1] Imagine that we have a number of documents stored in XML files, files that look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>

  <title>The Fellowship Of The Ring</title>
  <author>J. R. R. Tolken</author>
  <published>1954</published>

  <chapter>
     <title>A Long Expected Party</title>
     <content>When Mr. Bilbo Bagins of Bag End...</content>
  </chapter>

  <chapter>
     <title>A Shadow Of The Past</title>
     <content>The talk did not die down...</content>
  </chapter>

  <!-- ect -->
</document>
```

Now, the whole idea of an XML file is that the data is easily accessible and easy to manipulate. We go to the trouble of adding `<title>` and `<chapter>` tags to make it painless to extract data from the file. The XML world has defined XSLT, a language for doing this kind of thing, but XSLT is a complex technology in itself. For simple jobs a Ruby programmer will be happier with a little Ruby script. Fortunately, XML processing is also easy to do in Ruby. Here, for example, is a script that can read an XML file like the one above and tell you who the author is:

```
#!/usr/bin/env ruby

require "rexml/document"
```

1. Try as we might.

```
File.open( 'fellowship.xml' ) do |f|
  doc = REXML::Document.new(f)
  author = REXML::XPath.first(doc, '/document/author')
  puts author.text
end
```

This script relies on Ruby's very easy-to-use REXML XML parsing library, especially its XPath facility, which allows you to navigate through the XML hierarchy with convenient strings like `'/document/author'`. Given the author finding code above, it is just a tiny step to a script that finds all of the chapter titles:

```
#!/usr/bin/env ruby

require "rexml/document"

File.open( 'fellowship.xml' ) do |f|
  doc = REXML::Document.new(f)
  REXML::XPath.each(doc, '/document/chapter/title') do |title|
    puts title.text
  end
end
```

It's also easy to come up with a script to fix the misspelling of *Tolkien*'s name:

```
#!/usr/bin/env ruby

require "rexml/document"

File.open( 'fellowship.xml' ) do |f|
  doc = REXML::Document.new(f)
  REXML::XPath.each(doc, '/document/author') do |author|
    author.text = 'J.R.R. Tolkien'
  end
  puts doc
end
```

This last example is a little more complex than the first two, but not much. It reads the whole file, changes the text of the '/document/author' element, and then prints out the entire modified XML document.

As easy as REXML is to use, there is a lot of redundant code in those three little scripts. Each one needs to require in the REXML library, open the input file, and do some XPath-based searching. If we were going to do a lot of this sort of thing, we would want to factor out all of this repeated code into some kind of common utility, one that will help us rip through XML:

```
require "rexml/document"

class XmlRipper
  def initialize(&block)
    @before_action = proc {}
    @path_actions = {}
    @after_action = proc {}
    block.call( self ) if block
  end

  def on_path( path, &block )
    @path_actions[path] = block
  end

  def before( &block )
    before_action = block
  end

  def after( &block )
    @after_action = block
  end

  def run( xml_file_path )
    File.open( xml_file_path ) do |f|
      document = REXML::Document.new(f)
      @before_action.call( document )
      run_path_actions( document )
      @after_action.call( document )
    end
  end
```

```
    def run_path_actions( document )
      @path_actions.each do |path, block|
        REXML::XPath.each(document, path) do |element|
          block.call( element )
        end
      end
    end
  end
```

The `XmlRipper` class is built around the `@path_actions` hash, which maps strings containing XPaths to Ruby code blocks. The idea is that you fill out `@path_actions` hash by calling the `on_path` method repeatedly. When you are done you call the `run` method, passing in the name of an XML file. The `run` method will open the XML file, find the bits of XML that match the XPaths, and fire off the associated code block for each match.

The `XmlRipper` class dramatically reduces the drudgery of writing those XML processing scripts. Here's our author and chapter title examples all rolled into one, translated into `XmlRipper`:

```
ripper = XmlRipper.new do |r|
  r.on_path( '/document/author' ) { |a| puts a.text }
  r.on_path( '/document/chapter/title' ) { |t| puts t.text }
end

ripper.run( 'fellowship.xml' )
```

We can also fix the author's name with very little ceremony:

```
ripper = XmlRipper.new do |r|
  r.on_path( '/document/author' ) do |author|
    author.text = 'J.R.R. Tolkien'
  end
  r.after { |doc| puts doc }
end

ripper.run( 'fellowship.xml' )
```

This last example uses the `XmlRipper` `after` method to supply a block that gets run after all the other XPath-based processing is completed. The `after` method provides a convenient place to print out the whole XML document after we are done modifying it.[2]

## Stepping Over the DSL Line

Although we didn't set out to create a new language, the `XmlRipper` scripts certainly have a very declarative, specialized language feel to them. This is the way that many Ruby internal DSLs are born: You set out to build a helpful class with a good API, and gradually that API gets so good that it forgets that it's just an API. This is also where many Ruby APIs finish, which is fine since there is nothing wrong with a really good, natural-feeling, almost DSL-style API. Sometimes, however, you want to go further. You might want to take the next step if you need to write a lot of scripts, if there are a lot of programmers who will need to use your utility, or if the folks using `XmlRipper` are less technical. So how do you push `XmlRipper` along?

One way that you can make the `XmlRipper` scripts more DSL-like is to get rid of the need to constantly refer to the new `XmlRipper` instance (the `r` parameter) inside of the block. You can simplify the code inside of the block by turning to a method common to every Ruby object, `instance_eval`. Pass `instance_eval` a block and, just like `call`, it will execute the block. The difference is that `instance_eval` changes the value of `self` as it executes the block: Say `some_object.instance_eval( block )` and the value of `self` will be `some_object` as the block executes. Thus, if you rewrite the `XmlRipper` `initialize` method to use `instance_eval`:

```
class XmlRipper

  def initialize(&block)
    @before_action = proc {}
    @path_actions = {}
    @after_action = proc {}
    instance_eval( &block ) if block
  end
```

2. If you look carefully at the `XmlRipper` class you will see that there is also a `before` method, which operates as the mirror image of `after`. Although I'm generally against adding features that aren't used simply because they "make sense," this one makes so much sense that I'll make an exception. For more on this kind of thing, see Chapter 31.

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

end
```

Then `self` will be equal to the new `XmlRipper` instance as the block evaluates.[3] Since the `on_path` and `before` methods are defined on the `XmlRipper` instance, you can drop the initialization block argument and simply call `on_path` and `after` directly:

```
ripper = XmlRipper.new do
  on_path( '/document/author' ) do |author|
    author.text = 'J.R.R. Tolkien'
  end
  after { |doc| puts doc }
end

ripper.run( 'fellowship.xml' )
```

Getting rid of all those pesky `r` parameters is a step forward in making the `XmlRipper` scripts more language-like, but it's not the only thing you can do. Notice how each `XmlRipper` script always starts with the `XmlRipper.new` line and ends with the `ripper.run` call. In an ideal world you would get rid of all that boilerplate code, cutting down the code that the user has to write to the chewy center:

```
on_path( '/document/author' ) do |author|
  author.text = 'J.R.R. Tolkien'
end
after { |doc| puts doc }
```

Once again, `instance_eval` comes to your rescue. If, instead of passing a block to `instance_eval` you feed it a string, `instance_eval` will evaluate the string as Ruby code. Here's a new, slightly rewritten version of the `XmlRipper` class that can read the script from a file:

3. If this seems confusing, consider that `instance_eval( &block )` is equivalent to `self.instance_eval( &block )` and `self` is the newly created `XmlRipper` instance.

```
class XmlRipper

  def initialize_from_file( path )
    instance_eval( File.read( path ) )
  end

  # Rest of the class omitted...

end
```

Now you can write a short script that will read and execute a file full of `befores`, `on_paths`, and `afters`:

```
ripper = XmlRipper.new
ripper.initialize_from_file( 'fix_author.ripper' )
ripper.run( 'fellowship.xml')
```

More realistically, you would probably want to pass in command-line arguments so that you could pass in the name of the file containing your XML-manipulating script first, followed by the XML file itself:

```
r = XmlRipper.new
r.initialize_from_file( ARGV[0] )
r.run( ARGV[1] )
```

Congratulations! You have just witnessed the birth of a new Ruby internal DSL: Ripper. Based very firmly in Ruby and yet with a declarative, XML-processing feel all its own, Ripper tries to get the best of both worlds. By building atop Ruby, you managed to avoid all the work of creating a complicated parser for a brand new language. Since Ripper sits atop Ruby, you don't have to worry about implementing `if` statements and comments and the thousand other things that a useful programming language has. Ruby supplies them all for free:

```
# Correct a common mistake

on_path( '/document/author' ) do |author|
  author.text = 'J.R.R. Tolkien' if author.text =~ /Tolken/
end
```

```
# Print out the whole document when done

after { |doc| puts doc }
```

Free is always a good price.

## Pulling Out All the Stops

Be aware that Ruby internal DSLs are more a state of mind than a single technology. By building a DSL, you're going all out to make it easy for your user to do whatever the DSL does. Any programming technique that makes the job easier, that makes the code clearer, is fair game. Think about the possibilities of the metaprogramming techniques of the last few chapters: You might, for example decide that it would be helpful if Ripper users could specify very simple XPaths as part of the method name, like this:

```
on_document_author { |author| puts author.text }
```

Definitely a job for `method_missing`:

```
class XmlRipper

  # Rest of the class omitted...

  def method_missing( name, *args, &block )
    return super unless name.to_s =~ /on_.*/
    parts = name.to_s.split( "_" )
    parts.shift
    xpath = parts.join( '/' )
    on_path( xpath, &block  )
  end
end
```

The `method_missing` implementation here catches any method call that starts with on_, turns the rest of the method name into a simple XPath like `'document/author'`, and works from there. This is exactly the magic method technique that we explored in Chapter 23.

## In the Wild

Once you get the hang of the internal DSL techniques, the inner workings of a lot of seemingly magic Ruby utilities becomes obvious. Take a typical RSpec file:

```
describe "Array#each" do
  it "yields each element to the block" do
    a = []
    x = [1, 2, 3]
    x.each { |item| a << item }.should equal(x)
    a.should == [1, 2, 3]
  end

  # Lots of stuff omitted
end
```

This is either a structured description of the behavior of arrays or it is a call to a method named `describe`, a call that passes in a string and a block. Inside the block there is a call to a method called `it`, a method that also takes a string and a block. It's all very pedestrian when you know what is going on.

Another superb example of a Ruby DSL is Rake. Here's a simple Rakefile:

```
task :default => [ :install_program , :install_data ]

task :install_data => :installation_dir do
  cp 'fonts.dat', 'installation'
end

task :install_program => [ :installation_dir ] do
  cp 'document.rb', 'installation'
end

task :installation_dir do
  mkdir_p 'installation'
end
```

Again, we either have four task definitions or four calls to the `task` method. One very elegant thing about Rakefiles is the use of the hash literal syntax to specify dependency relationships. This:

```
task :default => [ :install_program , :install_data ]
```

Is just a brilliant way of saying that the `:default` task depends on both the `:install_program` and `:install_data` tasks.

As slick as RSpec and Rake are in all their "push Ruby to the limit" glory, many real-world Ruby programs are satisfied with less-ambitious DSL-like APIs. We've already looked at the ActiveRecord model API with its superclass-generated table relationships:

```
class Book < ActiveRecord::Base
  has_many :authors
  belongs_to :publisher
end
```

Even more basic, but no less effective are ActiveRecord migrations.

```
class AddBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.string  :title
      t.integer :publisher_id
    end
  end

  def self.down
    drop_table :books
  end
end
```

It's just a class that defines a couple of class methods, `up` and `down`, but it's also a description of how to build—and tear down—a database table.

## Staying Out of Trouble

As useful and easy as internal DSLs can be, they do have their downsides. The first is that internal DSLs tend to produce really bad error messages. Think about what would happen if we made a mistake in a Ripper script, perhaps like this:

```
# Error: Note the missing do on the first line...

on_path( '/document/author' ) |author|
  author.text = 'Tolkien'
end

after { |doc| puts doc }
```

Clearly, we forgot the do in the call to on_path. Unfortunately, as Ripper stands right now, this is the less-than-illuminating error message that our little screwup will produce:

```
ripper.rb:6:in `instance_eval': (eval):5:
  syntax error, unexpected keyword_end,
    expecting $end (SyntaxError)
  from ripper.rb:6:in `initialize_from_file'
  from ripper_main.rb:4:in `<main>'
```

The problem is that we messed up the Ripper program, but the error messages are coming back to us in Ruby terms. We can improve on this by using yet another bit of `instance_eval` magic. The `instance_eval` method has an optional second parameter, one that will tell `instance_eval` where the code it is evaluating came from. Pass in the name of the Ripper file as this second parameter:

```
class XmlRipper

  def initialize_from_file( path )
    instance_eval( File.read( path ), path )
  end

  # Rest of the class omitted...

end
```

And at least the error will point the user to the right file:

```
ripper.rb:6:in `instance_eval': broken.ripper:5:
 syntax error, unexpected keyword_end,
    expecting $end (SyntaxError)
 from ripper.rb:6:in `initialize_from_file'
 from ripper_main.rb:4:in `<main>'
```

You should also keep in mind that however helpful your DSL, you may still want to use your classes as an ordinary API. This is fairly easy if you keep the language-oriented bits of your DSL separate from the business end of the code, the part that actually does things. This way you can use your code via the DSL, or you can use it as part of an ordinary Ruby program.

Finally, it's worth noting that the biggest danger with internal DSLs is not really a computing problem but a psychological one. It goes by the name *programmer over enthusiasm*. Creating an internal DSL can help you squeeze a lot of power and flexibility out of very little code. The trick is to keep squarely focused on solving your real problem and avoid getting carried away with building a really cool syntax. Could we, for example, take the whole "XPath as Ruby code" idea to its logical conclusion? Perhaps, instead of the simple `method_missing` based `on_chapter_title` thing that we did in our last `XmlRipper` example, we could map the entire XPath syntax onto some Ruby code. So instead of saying this:

```
on_path( '/document/author' ) { |author| puts author.text }
```

We could say something like:

```
on_path /document/author { |author| puts author.text }
```

We might do this with a clever combination of operator overloading (think about the possibilities of the division operator!) and `method_missing`. Could it be done? I doubt it. Consider that while the XPaths that we used in this chapter are extremely simple, you can say some XPath things guaranteed to make Ruby gag. Any attempt to map `document/*/title` or `@*` into some DSL-like Ruby expression is destined to end in tears.

The real question is whether this kind of thing is even worth trying. Whatever goodness you bestow on your users by eliminating the quotes around `'/document/author'` is going to be more than canceled out by the Rabbinical complexity of the internal DSL code you'll need to write to make it all work. You can only push the Ruby parser so far; if you need to go beyond that point, roll up your sleeves and start thinking about writing your own parser.

## Wrapping Up

Internal DSLs are one of the hallmarks of Ruby done right. You take advantage of the flexibility of the language to create support for solving a whole class of problems. You can package that support as either a friendly API or you can keep pushing it to the point where it really is a new, specialized little language.

As useful as internal DSLs are, they are not the whole story. Sometimes you need a DSL but you simply cannot fit the syntax of your DSL into the strictures of Ruby. In those circumstances you will need to build an external DSL, which is where this book goes next.