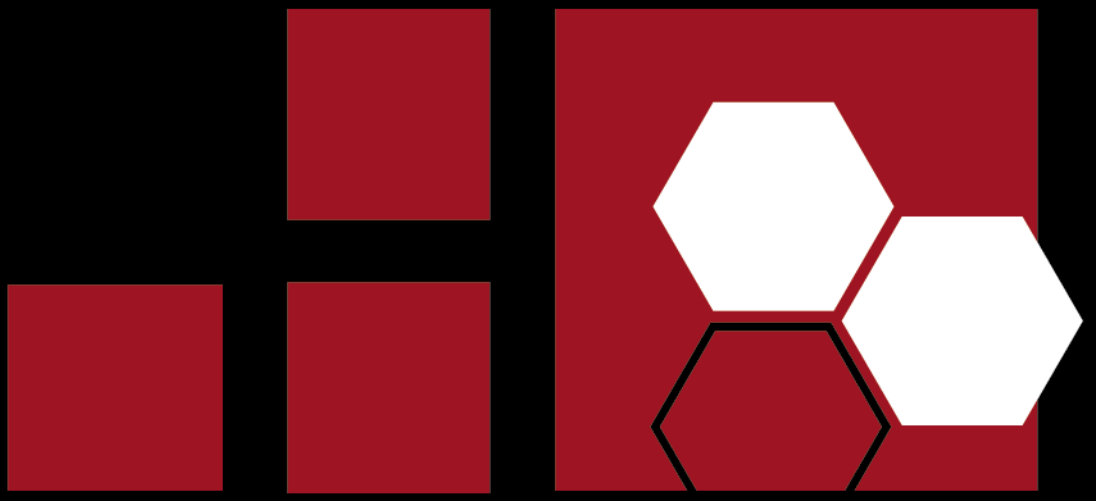

by Jim Gay

RUBY DSL HANDBOOK

Ruby DSL Handbook

Create Ruby code that speaks your language while avoiding metaprogramming hell

Jim Gay

Contents

Chapter 1

Updates to this book

- 16 Apr 2015
  - Happy birthday to my sister Mary!
  - Added example chapter for FactoryGirl
  - Added description of instance_exec to eval chapter
  - Added automatic scope switching section to antipatterns chapter
- 2 Apr 2015
  - Added links to more additional resources
  - Added chapter on External DSLs
  - Fixed incorrect return value in module helpers
- 20 Mar 2015
  - Added antipattern for switching between implicit and explicit interfaces
  - Added Additional resources chapter with Docile example
  - Updated Sharing Helper Code to show the creation of a ModuleName abstraction object

  - Fixed search and replace errors for module names

- 9 Mar 2015

  - Add chapter on DSLs without metaprogramming

- 3 Mar 2015

  - Better explained the need for constant names in Structure With Modules
  - Grammatical changes & rewording
  - Simplified and expandend the chapter of Sharing Helper Code
  - Added example chapter for Fattr

- 2 Mar 2015

  - Added chapter on handling arguments in generated methods
  - Added chapter on understanding eval

- 11 Feb 2015

  - Expanded the introduction of the has_many implementation
  - Expanded explanation of Selinum::WebDriver behavior
  - Added implicit vs explicit interface chapter
  - Added declarative setters chapter

- 5 Feb 2015

  - Expanded details of custom initializer

- 29 Jan 2015

  - Initial release

1.0.1 Learn how to quickly create Ruby code that speaks your language.

1.1 Thanks for purchasing

Ill release updates as new ideas, techniques, or support for approaches in Ruby DSL Handbook are added to the Ruby language. Your feedback is welcome. Please send your corrections, suggestions and ideas to jim@saturnflyer.com.

1.2 Licensing

1.3 What readers have to say:

"I learned some new aspects of Ruby I was able to apply the very same day" – Michael Buckbee

"With Jim's new book, Ruby DSL Handbook, he has once again tackled a difficult to understand problemspace in Ruby and brought it into readily understandable examples and explanations. His grasp upon the language comes out in force, in a way that even beginners can understand. Reading this book opened my eyes to different ways to tackle DSLs. It immediately had me wanting to explore different ways of reimplementing one of my own." – Jim Nanney

1.4 Advice before you read

My friend Kerri Miller had this to say about writing DSLs:

> is step 1 "DON'T" ?

Zach Briggs offered his insightful follow-up:

> Step 2: No, seriously. Don't

Their experience and sentiments are relevant to what you are about to read in this book. With metaprogramming, you can build the most fantastic of messes. It's important to consider whether or not you need to build a DSL in the first place. As Sandi Metz has said "Duplication is far cheaper than the wrong abstraction". You should be careful to think about why you want to create an abstraction and what you'll lose by creating it.

My goal with this manuscript is to provide you with the tools to compress ideas into simple statements and avoid losing information as we do with abstractions.

In writing this book, I'm making the assumption that you've considered your domain and have evaluated your needs before hacking away at it.

In the least, you should ask yourself first if you truly need to create a DSL.

Domain-specific language is a communication tool and can be used to great effect with your team members. It is particularly useful to those who are not code-literate and can provide a bridge to span the chasm between the code which implements a feature, and the understanding and requirements of what the feature is or does.

Consider this excerpt from an interview by Neal Ford of Rebecca Parsons and Martin Fowler:

> Neal: What would you have a developer tell a manager to convince them to allow some of these techniques?
>
> Rebecca: Two major benefits come to mind that apply in different situations. The first involves decreasing overall implementation

time (when one includes user testing) by increasing the effectiveness of the communication of requirements. If the problem has been properly communicated, the probability that the system does "the right thing" increases significantly, and the problem of having to rewrite software simply because the requirement was misunderstood decreases dramatically.

Martin: This is the brass ring of DSLs—the ability to open up a deep collaboration between programmers and domain experts. Often this is expressed as saying domain experts will write programs in DSLs. My sense is that wont be the case most of the time and that the more realistic point is where domain experts can read the DSLs. Readability is enough to unlock the key communication benefits.

Rebecca: The second benefit involves longer term maintenance. A system that is easier to understand is easier to maintain and evolve. Complex business logic buried in code is hard to understand. Putting that complex business logic in a language suited to it makes the job of understanding what the system really does simpler.

Communication is the key aspect of building a useful DSL. Begin your development through conversation with those who best understand the domain. Be prepared to change the DSL based upon your conversations. It is your job to write code which implements the language developed through discussion of your problem domain.

Use the code samples provided herein to create guideposts for your DSL. Creating restrictions in your code on what must, can't, or should be difficult to do will direct your development toward systems better designed for your needs.

1.5 Make it look like that

A good first step in preparing your DSL is to come up with the code that you want to use.

You'll need to think through what each part of your code will mean. Will you use Strings for particular information and Symbols for others? Or perhaps you'll use declarative methods to set values. Maybe you'll want the freedom to use procedural code to process information into an Array of values.

Providing guidance for exactly what to do requires knowledge of not only the domain for which you're creating a language, but also knowledge of the people who will use it and how they'll work together.

Before tackling the code to implement a DSL, try multiple different approaches to the DSL itself. What will it look like? What are the trade-offs between the different ways you could use it?

A good example to consider is the DSL for Rails request routing. When Rails began, your application's **`config/routes.rb`** file would look something like this:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :users
  map.resources :posts do |post|
    post.resources :comments
  end
end
```

Eventually the opinions of the maintainers of Rails changed and so did the DSL for routing:

```
Rails.application.routes.draw do
  resources :users
  resources :posts do
    resources :comments
  end
end
```

Although this particular change happened after several years and with several changes in-between, you should consider the possible differences for how you prefer to handle your DSL before you begin implementation. What do you think of the first style for routing? What is good or bad about it particularly when compared to the second?

This book is a guide to use either once you have already made decisions about how to build your DSL, or will help you make decisions about choosing one approach over another.

Chapter 2

Getting started quickly

Building and using DSLs in Ruby can be an amazing experience or a nightmare of metaprogramming hell. In this first chapter I want to give you the tools to get going quickly and in a way that wont lead to ruin. Lets dive in.

We want to create methods which do some code creation for us. Determining exactly what you want to do is obviously up to you and your project, but heres how to wrap your head around your task.

We'll start by creating a class to represent a web page and we want to specify that it has many parts of the page that we call features.

```
class Page
  has_many :features
end
```

This code won't work the way it is. It will raise an error:

```
NoMethodError: undefined method `has_many' for Page:Class
```

That's easy to overcome. But what we want this to do is create methods for us so that we can better reflect the concept of what a **`feature`** of a page is.

Let's discuss what that can be. For this domain, we'll be working with a representation of a web page and we'll use the **`Selenium::WebDriver`** library to traverse and interact with the HTML elements.

We'll interact with, count, compare, and do any number of other things by declaring that our page **has_many :features**.

We're about to look at a lot of code. The next code sample might be confusing at first but follow along with what makes sense to you and skip what doesn't. We'll walk through each part together.

We're going to use the argument provided to **has_many** to create a module containing the behaviors that we want. The methods generated by using **has_many** will be defined on that module, and it will then be mixed into our class.

Here's how we can make the above code work with a single class method:

```
class Page
  def self.has_many(name)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = 'HasMany' + name_to_module
    begin
      method_module = self.const_get(mod_name)
    rescue NameError
      method_module = Module.new
      self.const_set(mod_name, method_module)
      include method_module
    end
    line_no = __LINE__; method_defs = %{
      # define desired methods as an interpolated string here
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end

  has_many :features
end
```

This code will give you what you'll need to get started but as you can see in the comment "define desired methods. . . ", we still haven't defined any methods yet.

We'll get to that in a minute, but the first problem is to move this to a place that will be more manageable. Defining class methods is a great first step in creating your DSL but that will forever tie the implementation to this specific class.

You may find that want your code to remain as a class method. This can be useful if your are creating a DSL to alter the way subclasses of a class behave,

but it's a good idea to begin by understanding how to manage modules first.

Let's move our implementation to a module:

```
module HasMany
  def has_many(name)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = 'HasMany' + name_to_module
    begin
      method_module = self.const_get(mod_name)
    rescue NameError
      method_module = Module.new
      self.const_set(mod_name, method_module)
      include method_module
    end
    line_no = __LINE__; method_defs = %{
      # define desired methods as an interpolated string here
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end
end

class Page
  extend HasMany

  has_many :features
end
```

With this change, we can handle the method generation in the **`HasMany`** module and keep the code related to declaring what the **`Page`** *has* within the **`Page`** class.

The last bit of work we need to do is to write the code to define the instance methods that we'll need to use instances of this **`Page`** class.

Inside the **`has_many`** method we created above, there is a string called **`method_defs`** which, as you would likely expect, will contain our method definitions.

This **`Page`** class creates objects backed by **`Selenium::WebDriver`**. Let's initialize our pages and include a **`Selenium::WebDriver`** object called **`driver`**.

```
class Page
  def initialize(*args)
    # code to handle any arguments omitted...
```

```
    @driver = Selenium::WebDriver.for :firefox
  end
  attr_reader :driver
end
```

Your class might initialize in a different way. In this example, however, we're setting an instance of **`Selenium::WebDriver`** to **`@driver`** and setting a method to access that later using **`attr_reader :driver`**.

This will allow our pages to find HTML elements with a **`find_elements`** method.

```
line_no = __LINE__; method_defs = %{
  def #{name}
    driver.find_elements(:css, '.#{name}')
  end
}
```

What will happen with this code is that Ruby will parse this and generate something like this:

```
line_no = __LINE__; method_defs = %{
  def features
    driver.find_elements(:css, '.features')
  end
}
```

It's much clearer to see, now, that a method is created using the name we provided to **`has_many`** called **`features`** and will tell the **`driver`** to find elements using the same **`name`** argument.

With just this in your toolbelt, you should be able to get going quickly. If you you need to add more methods, merely alter the string to provide them:

```
line_no = __LINE__; method_defs = %{
  def #{name}
    driver.find_elements(:css, '.#{name}')
  end

  def #{name}_texts
```

```
    #{name}.map(&:text)
  end
}
```

Now you've dynamically created two methods: **features** and **features_texts** merely from using this code in your **Page** class:

```
class Page
  extend HasMany

  has_many :features
end
```

There are some stumbling blocks to overcome, depending on what you want to do. We'll see how to overcome these in the following chapters.

Chapter 3

Structure With Modules

The code we just wrote had quite a lot going on but there's good reason for doing it the way we did.

Before jumping into the code, let's discuss goals of the strategy.

We want our DSL to be non-invasive so that we may rely on the benefits of inheritance. The method we created manages constructing the name of a module and either finding or creating it. It does a bit more, but we'll focus on those parts right now.

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module
  begin
    method_module = self.const_get(mod_name)
  rescue NameError
    method_module = Module.new
    self.const_set(mod_name, method_module)
    include method_module
  end
  line_no = __LINE__; method_defs = %{
    # define desired methods as an interpolated string here
  }
  method_module.module_eval method_defs, __FILE__, line_no
end
```

What we want to be able to do when using our DSL is override it in a way that is convenient and flexible.

Here's the structure for our method lookup:

intended message -> object class -> module containing methods

We want the module to contain the methods so that they are available to be overriden.

Here's an example of how this can be powerful. In our previous examples, we changed the `has_many` methods which were generated to provide a `features_texts` method which gathered all the `features` from a page and their text. What would we do if we found that the implementation of that method wasn't exactly what we needed?

If our `features_texts` method included text but the page from where it pulled them also included excess whitespace, we'd get back the text we want with a lot of padded blank spaces.

Fortunately for us, the structure of our inheritance heirarchy allows for some easy overrides.

```
has_many :features

def features_texts
  super.map(&:strip)
end
```

We can rely on calling the `super` keyword to get the content from the originally implemented method of the same name: the `features_texts` method defined in the module created by `has_many`. Once we use the existing method, we can manipulate its output by stripping the excess whitespace.

Now that we've seen the benefits of the structure, let's pore over what's happening when we use `has_many`.

The `has_many` method accepts a single argument: `name`. The way we've used this argument is to provide a symbol with downcased text: `has_many :features`.

Because we want to create a module to hold our methods, it's useful to provide a name for that module.

When we create a new module with `Module.new` we can see a representation of it as something like this: `#<Module:0x007f901905da88>`. If we were to look at the ancestors of a class which includes a module like this it would look like this:

```
class Page
  include Module.new
end
Page.ancestors # => [Page, #<Module:0x007f901905da88>, Object, Kernel, BasicObject]
```

That list of ancestors can be confusing when we're trying to understand the composition of the `Page` class or it's objects. What methods does that anonymous module contain? What features does it provide?

Because anonymous modules can lead to more questions, we can use a simple solution by just assigning it to a constant. We'd much rather see something informative like this:

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

Ruby requires that constants begin with a capital letter. So we take the argument provide to `has_many` and use it to create a module name

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
```

This creates a string variable called `name_to_module` which is a camel case version of the `name` argument. The argument is converted to a string, then underscores are removed and their following letters are capitalized. This will take an arument like `:cute_kids` and provide a string of `"CuteKids"`.

The next step completes the name of the module:

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module
```

If we provide `:features` then the `mod_name` variable will become `"HasManyFeature`
While the naming of the module isn't crucial to our code, it'll be important for other uses of the same style of metaprogramming. We'll get to that later.

We next check for a module using that name to use it if it is already defined.

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module

    method_module = self.const_get(mod_name)
```

If this code doesn't work and Ruby can't find the constant with the name we've set to **mod_name**, this will raise a **NameError**. We want to rescue from that exception and create the module.

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module
  begin
    method_module = self.const_get(mod_name)
  rescue NameError
    method_module = Module.new
    self.const_set(mod_name, method_module)

  end
```

Our **has_many** DSL is designed to create instance methods for our classes. We want to include the module it creates so that methods defined in the module are added as instance methods in the class.

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module
  begin
    method_module = self.const_get(mod_name)
  rescue NameError
    method_module = Module.new
    self.const_set(mod_name, method_module)
    include method_module
  end
```

In the previous chapter we discussed the purpose of the string of method definitions. We won't go over it again here, but remember that the strings are generated to be what Ruby code we would write if we were defining the methods ourselves. The last thing to do after processing the method definitions

is to have the module that we created evaluate the string and turn it into real Ruby methods.

```
def has_many(name)
  name_to_module = name.to_s.split('_').map(&:capitalize).join
  mod_name = 'HasMany' + name_to_module
  begin
    method_module = self.const_get(mod_name)
  rescue NameError
    method_module = Module.new
    self.const_set(mod_name, method_module)
    include method_module
  end
  line_no = __LINE__; method_defs = %{
    # define desired methods as an interpolated string here
  }
  method_module.module_eval method_defs, __FILE__, line_no
end
```

We tell the **method_module** module to evaluate the string with **module_eval**. After that, the methods are created and we have instance methods available in our **Page** class or wherever else we've applied this **HasMany** module.

When adding methods to a module using **module_eval** we can take advantage of the two other arguments that it accepts. The first argument is a block or in our case a string, and then a reference to the current file and a line number.

Ruby provides **__FILE__** to easily get a reference to the currently running script and **__LINE__** to access the currently processed line. The timing of the use of **__FILE__** doesn't really matter, but the timing for **__LINE__** does.

Ruby will use the third argument in **module_eval** to provide information in the stack trace in the case of an exception raised by our code. This argument tells Ruby to begin processing line numbers from where this method is called. If we were to skip setting the **line_no** variable, and instead did this:

```
method_module.module_eval method_defs, __FILE__, __LINE__
```

We might get some interesting or perhaps just plain confusing reports from our exceptions providing incorrect line numbers as the location of the error.

We've looked at creating methods not on the class we're affecting, but in a module which includes it. By doing this, we can add behavior to a class and preserve our ability to override it with our own methods.

In the next chapter, we'll take a look at alternative approaches to handling the generated methods.

Chapter 4

Heredocs and method definitions

An alternative approach to creating your methods is to use heredocs.

```ruby
module HasMany
  def has_many(name)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = 'HasMany' + name_to_module
    begin
      method_module = self.const_get(mod_name)
    rescue NameError
      method_module = Module.new
      self.const_set(mod_name, method_module)
      include method_module
    end
    method_module.module_eval <<-CODE, __FILE__, __LINE__
      def #{name}
        driver.find_elements(:css, '.#{name}')
      end

      def #{name}_texts
        #{name}.map(&:text)
      end
    CODE
  end
end
```

The sample code is very similar to the previous implementation using a string. The **<<-CODE** is a signal to Ruby that we are using a "here document"

or "heredoc". Heredocs are processed starting with the line after the marker and ending with the line beginning with the specified marker.

In the line above, the **-** character allows the closing heredoc marker to occur anywhere on the ending line. The **<<-CODE** marker allows indentation, whereas **<<CODE** would require that the closing marker begin in column 1:

```
module HasMany
  def has_many(name)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = 'HasMany' + name_to_module
    begin
      method_module = self.const_get(mod_name)
    rescue NameError
      method_module = Module.new
      self.const_set(mod_name, method_module)
      include method_module
    end
    method_module.module_eval <<CODE, __FILE__, __LINE__
      def #{name}
        driver.find_elements(:css, '.#{name}')
      end

      def #{name}_texts
        #{name}.map(&:text)
      end
CODE
  end
end
```

One benefit of heredocs is that they allow for string interpolation without resorting to escaping quotes.

In the next chapter we'll see an example of this technique combined with lambdas to handle the parsing of our strings and creation of the methods we need.

Chapter 5

Sharing Helper Code

The `has_many` method is a bit complicated. It processes an argument into a particular string format, finds or creates a module, then applies methods to that module.

We can simplify this by extracting some behavior into individual methods. Here's a simplified version of `has_many` which moves the management of finding, initializing, and including the relevant `HasMany` module into a separate location:

```
module HasMany
  def has_many(name)
    method_module = HasMany.module_for('HasMany', name, self)
    line_no = __LINE__; method_defs = %{
      # define desired methods as an interpolated string here
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end

  def self.module_for(prefix, name, klass)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = prefix + name_to_module
    begin
      mod = klass.send(:const_get, mod_name)
    rescue NameError
      mod = Module.new
      klass.send(:const_set, mod_name, mod)
      klass.send(:include, mod)
    end
    mod
  end
```

```
end
```

Now the **has_many** method can rely on using the **module_for** method to simplify and focus on it's own needs.

As we develop additional libraries, creating a pattern of initializing and including modules will become useful beyond this single use. It makes sense to move these features to a shared location:

```
module ModuleHelper
  module_function def module_for(prefix, name, klass)
    name_to_module = name.to_s.split('_').map(&:capitalize).join
    mod_name = prefix + name_to_module
    begin
      mod = klass.send(:const_get, mod_name)
    rescue NameError
      mod = Module.new
      klass.send(:const_set, mod_name, mod)
      klass.send(:include, mod)
    end
    mod
  end
end

module HasMany
  def has_many(name)
    method_module = ModuleHelper.module_for('HasMany', name, self)
    line_no = __LINE__; method_defs = %{
      # define desired methods as an interpolated string here
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end
end
```

Once we have the code moved we are free to make changes by either shrinking the procedure or breaking it up into methods. Because this **module_for** method depends upon the rescuing an exception for a missing constant name, breaking it into multiple methods would likely only add to the complexity. A simplification of the variables will suffice:

```
module ModuleHelper
  module_function def module_for(prefix, name, klass)
    mod_name = prefix + name.to_s.split('_').map(&:capitalize).join

    begin
      mod = klass.send(:const_get, mod_name)
    rescue NameError
      mod = Module.new
      klass.send(:const_set, mod_name, mod)
      klass.send(:include, mod)
    end
    mod
  end
end
```

If the complexity of the name processing feels overwhelming, we can move the behavior into an object:

```
module ModuleHelper
  module_function def module_for(prefix, name, klass)
    mod_name = ModuleName.new(prefix, name)

    begin
      mod = klass.send(:const_get, mod_name)
    rescue NameError
      mod = Module.new
      klass.send(:const_set, mod_name, mod)
      klass.send(:include, mod)
    end
    mod
  end

  class ModuleName
    def initialize(prefix, name)
      @prefix = prefix
      @name = name
    end

    def to_s
      @prefix + @name.to_s.split('_').map(&:capitalize).join
    end
    alias to_str to_s
  end
end
```

This change is one way to manage encapsulating the concept of a name. In

this case, it might boil down to personal preference when determining which approach to use.

The `ModuleName` class is a representation of a string value, so we implement the features that Ruby expects to have for strings.

The `const_set` method expects to work with an object which can be coerced into a string. To coerce objects to strings, Ruby will call the `to_str` method. Without it, we'd get:

```
TypeError: Test is not a symbol or string
```

Creating an object to handle an idea like the name of a module can help us organize our code and our thoughts, but keep gotchas like this in mind when creating an abstraction.

5.1 Only sharing local behavior

It makes sense to create this separation so that this `ModuleHelper` may be used as a general utility. There may be times where creating this separation isn't worth any further effort.

Generalized functions can also work well as blocks of code which don't necessarily need to be extracted to a utility module.

The way the `has_many` method is structured, it expects to build a large string. We may find the need to create methods individually based upon additional processing.

We can take a tip from the implementation of the Fattr project which provides informative errors if the Ruby interpreter runs into syntax problems:

```
compile = ->(code){
  begin
    module_eval(code)
  rescue SyntaxError
    raise(SyntaxError, "\n#{ code }\n")
  end
}
```

If we create invalid code, we'll get the full code for the provided code back in the error message.

This `compile` lambda is a a useful utility that helps when defining methods individually:

```
module HasMany
  def has_many(name)
    method_module = ModuleHelper.module_for('HasMany', name, self)

    compile = ->(code){
      begin
        module_eval(code)
      rescue SyntaxError
        raise(SyntaxError, "\n#{ code }\n")
      end
    }

    code = <<-CODE
      def #{name}
        driver.find_elements(:css, '.#{name}')
      end
    CODE
    compile[code]


    code = <<-CODE
      def #{name}_texts
        #{name}.map(&:text)
      end
    CODE
    compile[code]
  end
end
```

As we can see in this code, we don't pass references for which module will receive the defined methods. In this case, it defines it on the current `self` object. This is something we're trying to avoid. The Structure With Modules chapter showed how we can avoid defining methods on the current class and allow us some flexibility to override default behaviors.

Here's how we can change this `compile` lambda to fit in with our approach to ensure the most flexible way to override behavior.

```
module HasMany
  def has_many(name)
    method_module = ModuleHelper.module_for('HasMany', name, self)

    compile = ->(code, line_no){
      begin
        method_module.module_eval(code, __FILE__, line_no)
      rescue SyntaxError
        raise(SyntaxError, "\n#{ code }\n")
      end
    }

    line_no = __LINE__; code = <<-CODE
      def #{name}
        driver.find_elements(:css, '.#{name}')
      end
    CODE
    compile[code, line_no]
  end
end
```

The **compile** lambda which handles any **SyntaxError** exception is a helpful tool for aiding in the process of **developing** the library whereas ensuring that **module_eval** receives the proper line number is a helpful tool for providing the end user better details about errors in code **using** the library.

Chapter 6

Creating a custom initializer

Common and repetitive tasks are ripe for moving into a DSL and often Ruby developers find themselves wanting to take care of initialization and setting of accessor methods.

The following example is a modified version of a custom initializer from the Surrounded project.

The goal of the custom initializer is to allow developers to simplify or shorten code like this:

```
class Employment
  attr_reader :employee, :boss
  private :employee, :boss
  def initialize(employee, boss)
    @employee = employee
    @boss = boss
  end
end
```

The above sample creates `attr_reader` methods for `employee` and `boss`. Then it makes those methods private, and next defines an initializer method which takes the same named arguments and assigns them to instance variables of the same name.

This code is verbose and the repetition of the same words makes it harder to understand what's going on at a glance. If we understand the idea that we want to define an initializer which also defines private accessor methods, we can boil that down to a simple DSL.

This is far easier to type and easier to remember all the required parts:

```
class Employment
  initialize :employee, :boss
end
```

There is one restriction. We can't provide arguments like a typical method. If we tried this, it would fail:

```
  initialize employee, boss
  # or
  initialize(employee, boss)
```

Ruby will process that code and expect to find **`employee`** and **`boss`** methods which, of course, don't exist. We need to provide names for what *will* be used to define arguments and methods. So we need to stick with symbols or strings.

Let's look at how to make that work.

Our first step is to define the class-level **`initialize`** method.

```
class Employment
  def self.initialize()

  end
end
```

Because we're creating a pattern that we can follow in multiple places, we'll want to move this to a module.

```
module CustomInitializer
  def initialize()
  end
end

class Employment
  extend CustomInitializer
end
```

Now we're setup to use the custom initializer and we can use it in multiple classes.

Because we intend to use this pattern in multiple places, we want the class-level **`initialize`** method to accept any number of arguments. To do that we can easily use the splat operator: **`*`**. Placing the splat operator at the beginning of a named parameter will treat it as handling zero or more arguments. The parameter **`*setup_args`** will allow however many arguments we provide.

The next step is to take those same arguments and set them as **`attr_reader`**s and make them private.

```
module CustomInitializer
  def initialize(*setup_args)
    attr_reader(*setup_args)
    private(*setup_args)

  end
end
```

With that change, we have the minor details out of the way and can move on to the heavy lifting.

As we saw in Structure With Modules we want to define any generated methods on a module to preserve some flexability for later alterations. We only initialize Ruby objects once; since we're defining the initialize method in a special module, it doesn't make sense for us to check to see if the module already exists. All we need to do is create it and include it:

```
module CustomInitializer
  def initialize(*setup_args)
    attr_reader(*setup_args)
    private(*setup_args)

    initializer_module = Module.new
    line = __LINE__; method_module.class_eval %{

    }, __FILE__, line
    const_set('Initializer', initializer_module)
    include initializer_module
  end
end
```

After we set the private attribute readers, we created a module with **`Module.new`**. We prepared the lines to evaluate the code we want to generate, and then we gave the module a name with **`const_set`**. Finally we included the module.

The last step is to define our initialize instance method, but this is tricky. At first glance it might seem that all we want to do is create a simple method definition in the evaluated string:

```
line = __LINE__; method_module.class_eval %{
  def initialize(*args)

  end
}, __FILE__, line
```

This won't work the way we want it to. Remember that we are specifying particular names to be used for the arguments to this generated method in our class-level **`initialize`** using **`employee`** and **`boss`** as provided by our **`*setup_args`**.

The change in scope for these values can get confusing. So let's step back and look at what we want to generate.

In our end result, this is what we want:

```
def initialize(employee, boss)
  @employee = employee
  @boss = boss
end
```

Our **`CustomInitializer`** is merely generating a string to be evaluated as Ruby. So we need only to look at our desired code as a generated string. With the surrounding code stripped away, here's what we can do:

```
%{
  def initialize(#{setup_args.join(',')})
    #{setup_args.map do |arg|
      ['@',arg,' = ',arg].join
    end.join("\n")}
  end
}
```

The `setup_args.join(',')` will create the string "employee, boss" so the first line will appear as we expect:

```
  def initialize(employee,boss)
```

Next, we use `map` to loop through the provided arguments and for each one we complile a string which consists of "@", the name of the argument, " = ", and the name of the argument.

So this:

```
  ['@',arg,' = ',arg].join
```

Becomes this:

```
  @employee = employee
```

Because we are creating individual strings in our `map` block, we join the result with a newline character to put each one on it's own line.

```
  %{

    #{setup_args.map do |arg|

    end.join("\n")}

  }
```

Here's our final custom initializer all the pieces assembled:

```
module CustomInitializer
  def initialize(*setup_args)
    attr_reader(*setup_args)
    private(*setup_args)

    method_module = Module.new
    line = __LINE__; method_module.class_eval %{
```

```
      def initialize(#{setup_args.join(',')})
        #{setup_args.map do |arg|
          ['@',arg,' = ',arg].join
        end.join("\n")}
      end
    }, __FILE__, line
    const_set('Initializer', mod)
    include mod
  end
end
```

Chapter 7

Implicit and Explicit code blocks

There are two styles of working with blocks where you explicitly work with an object provided as a block argument, or where you implicitly work with the object and there are no block arguments.

Here's an example of the different approaches:

```
# Explicit block objects
Sven.configue do |reindeer|
  reindeer.likes_carrots = true
  reindeer.better_than_people = true
end

# Implicit block objects
Olaf.configure do
  likes_warm_hugs = true
end
```

The first is explicit and the second is implicit. Explicit style is more verbose, but implicit style carries a heavy weight of what can be confusing scoping problems. Opt for explicit style first. Let's take a look at why...

7.1 Using yield

Possibly the simplest way to use a block is to yield the block and an object (or 2, or more) to the block as arguments. Here's an implementation of `Sven.configure`:

```
module Sven
  def self.configure(&block)
    block.call(self)
  end

  class << self
    attr_accessor :likes_carrots, :better_than_people
  end
end
```

The code above creates the `configure` method and also creates accessors for the attributes we care about. An altertative implementation of `configure` could use `yield`:

```
module Sven
  def self.configure
    yield self
  end
  # additional code omitted...
end
```

Every method in Ruby will accept a block argument. The self-documentating nature of creating an explicit block argument helps create an early signal to other developers that a block is expected. Although I don't specify `&block` exclusively and do use `yield`, I prefer to use `&block` for its clarity.

7.2 Yielding to an instance

Creating an implicit connection between the block and its receiver can be done with `instance_eval`:

```
module Olaf
  def self.configure(&block)
    instance_eval(&block)
  end

  class << self
    attr_accessor :likes_warm_hugs
  end
end
```

Using `instance_eval` evaluates the block on the current `self` object. An alternative to our implementation above would be to use `self.instance_eval(&block)`.

But we have a problem. Our `configure` doesn't set the `likes_warm_hugs` value like we want it to.

```
Olaf.configure do
  likes_warm_hugs = true
end
```

When Ruby evaluates the above code, it sees the `likes_warm_hugs` and sets it to true, but it sets it as a local variable. If we go to check for the value afterward, we'll find it was never set:

```
Olaf.likes_warm_hugs #=> nil
```

That's not helpful. So what happened?

In order to get Ruby to do the right thing, we need to provide a receiver to the `likes_warm_hugs` line so that it will attempt to use the accessor method that we created.

```
Olaf.configure do
  self.likes_warm_hugs = true
end
```

Certainly this is neither what we intend to do, nor will it serve as a clear communication tool. Non-technical readers of this code are likely to stumble

over this strange `self` word. Because of this need to specify the receiver, implicit interfaces lend themselves to declarative setters which help to avoid explicit receivers. We'll explore declarative setters in the next chapter, for now we can solve this problem by changing the way we set the value.

```
module Olaf
  # configure method omitted...

  def self.likes_warm_hugs
    @likes_warm_hugs = true
  end
end

Olaf.configure do
  likes_warm_hugs
end
```

7.3 Using additional information configure your objects

Problems with scoping can creep into your code when using implicit interfaces. Here's an example of using additional data to make decicions in the code and how it affects what can and can't be done.

With an explicit interface, implemented as we have with `block.call(self)` we can access information outside of our block.

```
people = ['Elsa','Hans']

Sven.configure do |reindeer|
  reindeer.likes_carrots = true
  unless people.include?('Anna')
    reindeer.better_than_people = true
  end
end
```

The code inside that `configure` block has access to the array of `people` and can use it as needed.

What happens with external objects using an implicit interface implemented with **`instance_eval(&block)`**?

```
marshmallow = OpenStruct.new(angry: true)

Olaf.configure do
  unless marshmallow.angry
    likes_warm_hugs
  end
end
```

When we run this code we find our problem:

```
NoMethodError: undefined method `angry' for nil:NilClass
```

The block for **`configure`** doesn't have access to the variables created outside of it. The **`marshmallow`** object appears to Ruby to be **`nil`** inside the block. We'll walk through using **`instance_eval`** in detail later but this exercise illustrates the rigidity that using it can introduce.

Using an explicit interface implemented with **`block.call`** or **`yield`** gives us code making scoping problems easier to avoid.

When designing your DSL interface, it would be best to avoid implicit code if you my have the need to access data external to your block. If the users of your DSL will have a clear understanding of the scope required, then an implicit interface can remove the noise related to working with Ruby and make your DSL feel more like working with the domain problem.

Chapter 8

Declarative Setters

In our discussion of implicit and explicit interfaces, we saw this sample code:

```
# Explicit block objects
Sven.configue do |reindeer|
  reindeer.likes_carrots = true
  reindeer.better_than_people = true
end

# Implicit block objects
Olaf.configure do
  likes_warm_hugs = true
end
```

Unfortunately, as we saw in the implicit vs. explicit discussion, this doesn't work the way we want it to.

To prevent Ruby from storing `likes_warm_hugs` as a local variable, we can treat it as a declaration of the value rather than a procedure which sets it. Here's how this changes our configuration:

```
Olaf.configure do
  likes_warm_hugs true
end
```

In order to make this work we'll need to forego our `attr_accessor` and define the method ourselves:

```
module Olaf
  # configure method omitted...

  def self.likes_warm_hugs(value)
    @likes_warm_hugs = value
  end
end
```

This change allows our configuration to work. We're still faced with a problem, however.

Accessing this value requires that we either create another method to make the query for us, or we need to overload the behavior of this setter. Here's an example use of an implementation we'll review:

```
Olaf.configure do
  likes_warm_hugs true
end
# query methods
Olaf.get_likes_warm_hugs # => true
Olaf.likes_warm_hugs? # => true
# or overload the behavior of the setter
Olaf.likes_warm_hugs # => true
```

We can opt to create query methods to retrieve the set value

```
module Olaf
  # configure method omitted...

  def self.get_likes_warm_hugs
    @likes_warm_hugs
  end
end
```

When designing the interface we need to be careful about what we imply. A method called **`likes_warm_hugs?`** implies that it will return a boolean value: true or false. Some developers employ a hard rule that methods ending with a question mark ("?") must return an actual boolean, whereas others allow a truthy value. In other words, you can implement it to return a value which evaluates to true or false when put into an **`if`** (or other similar) statement.

A problem with returning truthy values is that code may be introduced which depends upon the value *not* being a boolean and subsequently becomes difficult to understand.

Here's an example of how that approach can confuse things:

```
module Olaf
  # configure method omitted...

  def likes_warm_hugs?
    @likes_warm_hugs
  end
end

Olaf.configure do
  likes_warm_hugs :sometimes
end

hug_setting = Olaf.likes_warm_hugs?
```

When *reading* the code storing the **`hug_setting`** variable we'd expect the value to be a boolean, but it isn't.

It's important to consider how we'll use information recorded by our DSL. Driving our conversation about the domain through the use of the information will help us make better decisions.

An alternative to providing a query method is to allow our **`likes_warm_hugs`** method to accept an argument and store it, or return the value as stored.

```
module Olaf
  # configure method omitted...

  def self.likes_warm_hugs(val=:not_set)
    if val == :not_set
      @likes_warm_hugs
    else
      @likes_warm_hugs = val
    end
  end
end

Olaf.configure do
  likes_warm_hugs true
end

Olaf.likes_warm_hugs # => true
```

This implementation employs the use of an unlikely value. If we are unlikely to use the value `:not_set` then we can make our code more readable by using it as a flag to return the value instead of setting it.

This allows us to use values which evaluate to false as legal values for our method.

```
Olaf.configure do
  likes_warm_hugs false
end
```

8.1 Set and unset values

An alternative to these is to treat our `likes_warm_hugs` method only as a setter where the value is set or unset.

```
module Olaf
  # configure method omitted...

  def self.likes_warm_hugs
    @likes_warm_hugs = true
  end
end

Olaf.configure do
  likes_warm_hugs
end
```

With this approach, we can declare that `Olaf` does indeed like warm hugs, and leave the values out of it. If we find that we need to query for them, we can leave that to an internal implementation requirement and create a private query method.

Chapter 9

Handling Arguments in Generated Methods

Let's take a look at creating methods which accept arguments.

Many of the examples you've read have used string interpolation with `class_eval` to create methods. This is convenient but can be a mind-bender when you need to create methods which also use string interpolation with their arguments. But it doesn't have to be.

Here's a small example.

Let's create a method which uses `Selenium::WebDriver` to search for elements on a page.

In the sample method we'll create `gifts_for` which will get a list of elements from an HTML page and filter based upon an argument that we provide.

We can use the method to get elements from different categories `gifts_for("mom")`, `gifts_for("dad")`, `gifts_for("kids")` or whatever other option there might be.

```
def gifts_for(whom)
  find_elements(css: ".gifts-for-#{whom}")
end
```

The `gifts_for` method searches through the HTML using CSS selectors where items have attributes containing information about the type of element it

is.

When defining this method ourselves, the string interpolation is easy to understand. When we use string interpolation to create the method, we need to be careful.

The following is an example which uses our **ModuleHelper** from the "Sharing Helper Code" chapter.

We're going to be generating this method so let's start with something similar to our **has_many** method from previous chapters. We'll say that the page **has_options** and while it could be anything, in our case we'll say it **has_options(:gifts)** to handle the gift elements on our page.

The class-level method **has_options** will accept a **name** argument which we'll set to **:gifts**.

That means that in order to create **def gifts_for(whom)** we'll want to generate a method using this string **"def #{name}_for(whom)"**

```
module Options
  def has_options(name)
    method_module = ModuleHelper.module_for('Options', name, self)
    line_no = __LINE__; method_defs = %{
      def #{name}_for(whom)
        find_elements(css: ".gifts-for-#{whom}")
      end
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end
end

class Page
  extend Options
  has_options(:gifts)
end
```

If this code were run, we'd see an error. The scope of the variables **name** and **whom** are actually at the same level.

Despite that it appears in this code that the **find_elements** arguments are inside the **#{name}_for** method definition, we're really only generating a string, and any cues to Ruby for interpolation are processed right away.

Taking a closer look, with all the excess code removed, we can see the problem:

```
%{
  def #{name}_for(whom)
    find_elements(css: ".gifts-for-#{whom}")
  end
}
```

When parsed, the **whom** argument is expected to exist when the string is being defined, not when the method it represents is being run.

One way to get around this is to compose the **find_elements** arguments inside the generated method:

```
%{
  def #{name}_for(whom)
    css_locator = [".gifts-for-",whom].join
    find_elements(css: css_locator)
  end
}
```

When we create the array **[".gifts-for-",whom]** it is done when the method is called. The **whom** variable will be processed when the argument is passed to the method.

Another, though less readable, option is to escape the generated characters for interpolation:

```
%{
  def #{name}_for(whom)
    find_elements(css: ".gifts-for-\#\{whom\}")
  end
}
```

I prefer to *not* use this latter style. The excess characters used to escape important characters used in later interpolation make this difficult to scan quickly and get a good understandnig of what's going on. While the same could be said about joining elements from an array as we did with the **css_locator** variable, I find it far less error-prone.

You should, however, keep the option for using escape charaters in mind. As you build and benchmark the performance of your project you may find

that the creation of an arry and subsequent joining of it's elements costs too much. At that point, the cost of reading escape characters may be less than the performance impact of the chosen approach.

Here's a look at the final result.

```
module Options
  def has_options(name)
    method_module = ModuleHelper.module_for('Options', name, self)
    line_no = __LINE__; method_defs = %{
      def #{name}_for(whom)
        # find_elements(".gifts-for-\#\{whom\}") # perhaps this may be necessary, but I'd avoid
        css_locator = [".gifts-for-",whom].join
        find_elements(css: css_locator)
      end
    }
    method_module.module_eval method_defs, __FILE__, line_no
  end
end
```

Reading the commented out portion takes me a bit longer and I'm less confident that I know what I saw. But the composition of the **`css_locator`** variable is very clear.

Chapter 10

Understanding _eval

When first approaching `instance_eval` and `class_eval` the method names can be confusing.

In the terminology we use to discuss Ruby code, we use a class to initialize an instance of that class. Inside each class we define instance methods to be available to the instances or class methods to be available to the class.

If we want to dynamically define instance methods for a class, we use `class_eval`, whereas defining class methods we might use `instance_eval`. Here's an example of this craziness:

```
class Person
end

Person.class_eval do
  def hello
    'hello'
  end
end

Person.instance_eval do
  def goodbye
    'goodbye'
  end
end

Person.new.hello # => "hello"
Person.goodbye # => "goodbye"
```

Believe it or not, this makes plenty of sense. Here's why...

When we create a class in Ruby, we have different ways to do it. You're familiar with the typical way:

```
class Person
end
```

But we can also create a class by initializing one from the **Class** class.

```
Person = Class.new
```

With this second example it's easy to see that **Person** is an instance of **Class**. We can define methods on this class as it is created.

```
Person = Class.new do
  def an_instance_method
    'used on instances of this class'
  end
end
```

When using **instance_eval**, Ruby will evaluate the provided string or block against the object receiving the **instance_eval** message.

Every object in Ruby responds to **instance_eval**, but only modules and classes respond to **class_eval** (and **module_eval**).

To define methods on the receiver, we use **instance_eval**:

```
Person.instance_eval do
  def class_level_method
    'this is a method for the Person class'
  end
end
Person.class_level_method # => "this is a method for the Person class"
Person.new.class_level_method # => NoMethodError
```

Because classes are factories which generate instances of themselves (instances which are not also classes) we can use **class_eval** to create methods for instances of that class (and not the class itself):

```
Person.class_eval do
  def instance_level_method
    'this is a method for instances of Person'
  end
end
Person.instance_level_method # => NoMethodError
Person.new.instance_level_method # => "this is a method for instances of Person"
```

Now, stepping back to our **`Class.new`** example, we can see that it is the equivalent of using **`class_eval`** on the initialized class:

```
Person = Class.new.class_eval do
  def an_instance_method
    'used on instances of this class'
  end
end
```

To illustrate the differences between **`instance_eval`** and **`class_eval`**, the following code shows the equivalent code using traditional ways of creating methods:

```
class Person
end

Person.class_eval do
  def hello
    'hello'
  end
end
# is equivalent to:
class Person
  def hello
    'hello'
  end
end

Person.instance_eval do
  def goodbye
    'goodbye'
  end
end
# is equivalent to:
class Person
  def self.goodbye
```

```
    'goodbye'
  end
end
```

10.1 The _exec sisters of _eval

The **instance_eval** and **class_eval** methods have a set of sister methods which behave similarly.

Modules and classes are able to execute a block of code in their context using **instance_exec** and **class_exec** (and **module_exec** of course). So what's the difference between these and their **_eval** sisters?

These methods allow you to provide a block of code for execution, and to provide arguments to the block.

```
class Person
  def initialize(name)
    @name = name
  end
  attr_reader :name
  attr_accessor :location
  private :location

  def outline(text)
    location.instance_exec(text) do |text|
      puts "at #{name} #{Time.now.strftime('%I:%M:%S %p')}: #{text}"
    end
  end
end
```

This implementation means that the **outline** method will have a block argument available for use with the related location.

```
class Place
  def name
    self.class.to_s
  end
end
class Office < Place; end
class Home < Place; end
```

```
class Restaurant < Place; end

home = Home.new
office = Office.new
restaurant = Restaurant.new

jim = Person.new('Jim')
jim.location = home
jim.outline("Getting started")
jim.outline("Cleaning up")
jim.location = office
jim.outline("Getting some work done")
jim.outline("Lunchtime!")
jim.outline("Closing down!")
jim.location = restaurant
jim.outline("Eating Dinner...")
```

This allows the code using this **`outline`** method to provide information from outside the context of the **`location`** and the block inside the **`outline`** has access to any methods on the **`location`** instance, such as **`name`**.

Take note: the bare **`exec`** method is entirely different from these that we've discussed. Unlike **`eval`** which behaves similarly to the other **`_eval`** methods, **`exec`** replaces the currently running Ruby process with the command you've provided to it.

Chapter 11

A DSL without metaprogramming

Almost all of the code samples in this book revolve around the techniques that you can use with Ruby's metaprogramming features.

Building a DSL doesn't require metaprogramming.

A DSL provides an abstraction for more complicated ideas. By creating a DSL, we simplify and exclude information by using names relevant to our domain.

Here's an example of a DSL used in a minitest file from my friend Ken Collins (creator of HomeMarks).

I've removed code specific to gathering information from the database so that we can focus on the parts relevant to what Ken created to handle his domain.

```
describe 'sorting' do
  it 'drags 3rd DevOps bookmark to 1st trashbox bookmark' do
    open_inbox
    open_trashcan
    bookmark_handle('DNSimple').drag_to bookmark_handle('PHP')

    within(trashbox_el) do
      # asserts
    end
    # asserts
    within(box_devops_el) do
```

```
      # asserts
    end
    # asserts
  end

end
```

The methods **open_inbox** and **open_trashcan** at the beginning of that test are simple descriptions of what happens when interacting with the HomeMarks interface. From the perspective of this test in the **it** block, we don't care specifically how **open_inbox** works. The method is an abstraction over the procedure required to interact with the system and to produce the desired result. In other words, if we have to click on one thing or twenty in order to get the inbox opened, the test only has an interest in the final state of the inbox.

Here's what the **open_inbox** method looks like:

```
def open_inbox
  click_toolbar_button 'Inbox & Trash'
  wait_for_inbox_to_appear
end
```

This method depends on three things:

1. **click_toolbar_button**
2. The page has an element with text **'Inbox & Trash'**
3. **wait_for_inbox_to_appear**

Each of those dependencies may have their own dependencies as well.

The purpose of creating this language for the HomeMarks domain is to do less thinking about the parts that don't matter. This test is concerned with the movement of elements from one location to another and that they properly appear.

Here's some of the methods which support the above test:

```
def toolbar_button(name)
  all('.toolbar .btn').detect { |a| a[:title] == name }
end

def click_toolbar_button(name)
  toolbar_button(name).click
end

def trashcan
  find '.trashcan'
end

def open_inbox
  click_toolbar_button 'Inbox & Trash'
  wait_for_inbox_to_appear
end

def open_trashcan
  trashcan.click
  wait_for_trashcan_to_appear
end
```

Were we to write thes pieces in the test, it would be muddy with multiple concerns of differing levels:

```
describe 'sorting' do
  it 'drags 3rd DevOps bookmark to 1st trashbox bookmark' do
    click_toolbar_button 'Inbox & Trash'
    wait_for_inbox_to_appear
    find('.trashcan').click
    wait_for_trashcan_to_appear

    # additonal test code omitted...
  end
end
```

We can see in the expanded test code how much noise there is. The noise of particular bits of text like `'Inbox & Trash'` or CSS classes with `'.trashcan'` in addition to commands to click or wait for an object to appear completely distract us from the purpose.

A DSL can be extremely simple, and yet powerful enough to help us focus on the important parts.

Each method we create in our programs has a meaning and purpose particular to its domain. While we might build systems which are concerned with

managing URL bookmarks, those systems will likely depend on a lower level DSL to better manage distracting information.

Keep your code simple with your DSL. Choose what information is important and which information is a distraction.

Looking at this sample code, you might think that you would do things differently. Perhaps we might want to use signifiers for elements on the page:

```
describe 'sorting' do
  it 'drags 3rd DevOps bookmark to 1st trashbox bookmark' do
    open(:inbox)
    open(:trashcan)

    # additonal test code omitted...
  end
end
```

Maybe in our own projects, we decide to name important elements using symbols. In this case, we could treat the important elements with **`:inbox`** and **`:trashcan`**.

The implementation of the **`open`** method might be difficult to manage:

```
def open(which)
  if which.to_s =~ /inbox/
    click_toolbar_button 'Inbox & Trash'
  else
    find(".#{which}").click
  end
  self.send("wait_for_#{which}_to_appear")
end
```

This looks difficult to understand.

If we were to go down this route for implementation of the DSL we can decide one of two things. Either the implementation of the DSL is too complex and we should move to individual methods like **`open_inbox`** and **`open_trashcan`**, or the implementation of the DSL points to difficulty with the system and we should alter that.

The choices we make help to inform our design of both our DSL and our system. Discomfort with any aspect of the code we right either points to that code being too difficult, or it points to our system being too difficult.

Chapter 12

External DSLs

Most of this book is about building *internal* DSLs. That is, the language that you create with an internal DSL relies on the syntax of the implementation language. Up till now, our language was merely an augmentation of Ruby.

External DSLs carry significant overhead in implementation because they support syntax that is often outside the bounds of the implementation language.

External DSLs require that you create a parser to help the computer understand the language you've created and turn it into executable code. This can be an enormous undertaking if you don't already have any experience doing so. Creating an external DSL requires that you define how a language works. You must define its structure and understand how exceptions might arise. You'll of course want to think about what the user of your DSL will read and understand how to work with your language.

There's quite a lot to think about for external DSLs, so we'll only dip our toes in the water.

12.1 Regular Expressions

Regular expressions are the easiest tool to grab when you want to parse some text.

We can create simple expressions in our own language, capture the parts we care about, and use them to do some work.

The way we write phone numbers is a good example of a simple external DSL that is easy to parse with regular expressions. We don't often consider phone numbers as a domain specific language but we do assign particular meaning to each part. The phone number representation has a particular structure which helps us better understand it as a whole.

Let's take an example number written like this: **(212) 555-1234**

We can create a regular expression to match this number

```
/\(\d{3}\)\s\d{3}-\d{4}/
```

This expression matches the sample number format exactly:

```
number = '(212) 555-1234'
exp = /\(\d{3}\)\s\d{3}-\d{4}/
exp.match(number) #=> #<MatchData "(212) 555-1234">
```

When given text which doesn't conform to our phone number format, we'll match nothing:

```
non_number = '212 555-1234'
exp = /\(\d{3}\)\s\d{3}-\d{4}/
exp.match(non_number) #=> nil
```

Before addressing alternative formats like that, let's change our regular expression so we can capture the individual parts of our phone numbers.

In a US phone number, the area code is often written in parentheses such as "(212)". The number "212" represents a geographic location to where phone systems will route a call. The second part of the number is called the prefix and is a three-digit group which originally represented the switch to which a phone line was connected ("555" in our example). And the final four digits, which are appended after a hypen ("-"), represent the line number.

Regular Expressions are a DSL for processing a repeating pattern in strings of characters. To capture individual pieces, we wrap the part we want with paretheses. If we want to check for an actual parethesis character, we use the

backslash (“") to escape it. We match digits with “d” and can provide a quantity in curly braces. So “d{3}” will match three digits.

We can change our expression to capter the numbers we care about: “212”, “555”, and “1234”.

```
number = '(212) 555-1234'
exp = /\((\d{3})\)\s(\d{3})-(\d{4})/
exp.match(number) #=> #<MatchData "(212) 555-1234" 1:"212" 2:"555" 3:"1234">
```

As you can see, the **MatchData** object has additional values which map to the phone number parts. We can access these parts a few ways but the simplest is using the **[]** method and the index as set in the **MatchData** object.

```
exp.match(number)[1] #=> "212"
```

Ruby allows us to use our own names for each of these parts. This helps our code to feel much more like the language we want to use:

```
number = '(212) 555-1234'
exp = /\((?<area_code>\d{3})\)\s(?<prefix>\d{3})-(?<line>\d{4})/
exp.match(number) #=> #<MatchData "(212) 555-1234" area_code:"212" prefix:"555" line:"1234">
```

This makes accessing our captures much easier to understand. We don’t need to care about the order or a specific number, we can get the data by name:

```
exp.match(number)[:area_code] #=> "212"
```

To address optional parenthesis, we can use the “?” metacharacter to specify that the parenthesis charaters may or may not be present.

```
exp = /\(?(?<area_code>\d{3})\)?\s(?<prefix>\d{3})-(?<line>\d{4})/
```

Accessing the data is much clearer with the named captures, but getting comfortable with the added noise in the regular expression can be difficult. We've only used the simple aspects of regular expressions in Ruby. If we create a DSL beyond something simple like phone numbers, this could become not just difficult to read, but difficult to debug if it isn't working properly.

12.2 Using Parslet

An alternative to a simple solution with regular expressions is to create a parser.

Let's take a look at Parslet to support a simple external DSL. For a very small case, let's create a language which allows us to add numbers. We'll write it like this

```
add 4, 5
```

We may want to add multiple numbers, so we'll allow that too:

```
add 8, 17, 28, 3
```

It's important to understand that the syntax structure will affect our program. Just like our regular expression approach to processing numbers, the code will expect things to be in a particular order.

First we'll create a representation of each of the parts we care about to see how Ruby will interact with the content provided by the DSL.

We will take each of the pieces, process them into their appropriate Ruby parts, and then evaluate them.

Each number will be created as an **`IntegerLiteral`** on which we'll call **`eval`**.

```
class IntegerLiteral
  def initialize(num)
    @int = num
  end
  attr_reader :int
```

```
  def eval
    int.to_i
  end
end
```

When we get to adding these numbers, we'll send them into an **Addition** object to handle the calculation:

```
class Addition
  def initialize(list)
    @args = list
  end
  attr_reader :args

  def eval
    args.map(&:eval).reduce(:+)
  end
end
```

The **Addition** objects contain a list of arguments. When the addition is evaluated, it will evaluate each of its arguments using the **eval** method, and then add them all together using **reduce(:+)**.

These are the individual parts which connect to the Ruby object world. The **IntegerLiteral** turns text for numbers into Ruby Integers, and the **Addition** taks a list of those **IntegerLiteral** objects and adds them together.

As our next step, we'll start creating our parser to make sense of our language and break it into named parts.

```
require 'parslet'

class Lang < Parslet::Parser
  # Single character rules
  rule(:space)      { match('\s').repeat(1) }
  rule(:space?)     { space.maybe }
  rule(:comma)      { str(',') >> space? }

  # Particulars
  rule(:integer)    { match('[0-9]').repeat(1).as(:int) }
  rule(:operator)   { match['a-z'].repeat(0) }
```

```
  # Grammar
  rule(:expression) { sum | integer }
  rule(:arglist)    { expression >> (comma >> expression).repeat }
  rule(:sum)    {
    operator.as(:sum) >> space >> arglist.as(:arglist) }

  root :expression
end
```

The `Lang` class inherits from the `Parslet::Parser` class and defines a set of rules. Each rule matches either an individual character or group of features from the DSL text.

The `:arglist` and `:sum` rules define the larger parts of the grammar. An `:arglist` can contain a list of comma separated expressions. The `:sum` is defined as the operator ("add" in our case) followed by a space and then an `:arglist`. The `:expression` is either a `:sum` or an `:integer`.

Lastly, we tell the parser to begin processing, starting at the `root`, with the `:expression` rule.

Our final step is to create the code to transform the parsed DSL content into the Ruby objects.

```
class Transformer < Parslet::Transform
  rule(:int => simple(:int))        { IntegerLiteral.new(int) }
  rule(
    :sum => 'add',
    :arglist => subtree(:arglist))  { Addition.new(arglist) }
end
```

This `Transformer` turns our DSL integers and functions into `IntegerLiteral` and `Addition` objects. By inheriting from `Parslet::Transform` we'll have a few special methods but in particular the most important here is that we'll use `apply` and `eval`:

```
transformer = Transformer.new
parser = Lang.new

syntax_tree = transformer.apply(parser.parse("add 3, 4"))

puts syntax_tree.eval
```

This last step takes the parser and the transformer we just created, breaks our DSL text into parts and assigns them to the appropriate Ruby classes. The last call to `eval` sets them in motion.

As you can see, there's a lot to do when handling an external DSL. Not only can creating a parser and compiler be a significant task, but coming up with the syntax and semantics of your DSL can be a challenge too.

External DSLs can help you focus on just the parts you need for your domain. Seeing your domain through the eyes of Ruby, and then only using an internal DSL, can cause confusion and create stumbling blocks over the syntax requirements of a language not designed specifically for your domain.

Chapter 13

Example: Direction

A small and little-used gem called Direction creates a tiny DSL for implementing Command/Query separation.

The intent of the DSL is to specify commands which forward to another object, but which always return the receiving object as the result of the command.

A common pattern of returning the receiving object can be difficult to enforce. Direction provides behavior similar to the Ruby standard library Forwardable.

Here's an example from Direction:

```
class Person
  extend Direction

  command [:print_address] => :home

  attr_accessor :home
end

class Home
  def print_address(template)
    template << "... the address.."
  end
end

template = STDOUT
person = Person.new
person.home = Home.new
```

```
#commands won't leak internal structure and return the receiver of the command
person.print_address(template) #=> person
```

This sample shows that a **Person** can hold a reference to **home**. The **command** creates a method which tells the **home** object to **print_address** with the provided arugment. Rather than returning the result, the **print_address** method on **person** returns the **person** object itself.

If we expanded the methods marked as commands, it would look like this:

```
class Person
  def print_address(template)
    home.print_address(template)
    self
  end
end
```

Although the implementation is simple, the DSL allows for the pattern to be easily implemented and the return value to be enforced.

Direction provides both **command** and **query** methods to specify that the result of the first will be the receiving object and the result of the second will be whatever the result of the forwarded message is.

Let's look at how that plays into the creation of the code supporting the DSL:

```
module Direction
  # Forward messages and return self, protecting the encapsulation of the object
  def command(options)
    Direction.define_methods(self, options) do |command, accessor|
      %{
      def #{command}(*args, &block)
        #{accessor}.__send__(:#{command}, *args, &block)
        self
      end
      }
    end
  end
  #...
```

The **command** method manages creating a string representation of methods and ensures that the return value of the generated method is the current object.

The library moves the heavy lifting of defining the methods into the `Direction.defin` method. This leaves the `command` (and subsequently `query`) methods to focus on what is different about them.

Let's look at the `query` method:

```
module Direction
  #...
  # Forward messages and return the result of the forwarded message
  def query(options)
    Direction.define_methods(self, options) do |query, accessor|
      %{
      def #{query}(*args, &block)
        #{accessor}.__send__(:#{query}, *args, &block)
      end
      }
    end
  end
  #...
```

The `query` method creates a string for the method definition which has no particular return value.

```
module Direction
  # ...
  def self.define_methods(mod, options)
    method_defs = []
    options.each_pair do |method_names, accessor|
      Array(method_names).map do |message|
        method_defs.push yield(message, accessor)
      end
    end
    mod.class_eval method_defs.join("\n"), __FILE__, __LINE__
  end
end
```

The `define_methods` method accepts a module and hash of options used to define the command or query methods.

A `method_defs` array is created to accumulate each method definition. The code then iterates over the `options` argument and yields control to the provide block for each method name and accessor name.

For more about why you would structure your object relationships like this, read Clean Ruby which explains this pattern of East-oriented Code.

Chapter 14

Example: Fattr

Ara Howard created a project called Fattr which implements a common desire among Ruby developers: a concise way to specify attributes of an object with default values. The project's describes itself as: a "fatter attr" for ruby.

Often developers want to specify that a class of objects will have `attr_reader` or `attr_accessor` but they want to add something else.

Here's an example of the problem. In order to take something as simple as `attr_accessor :color` and provide a default value, you'd need to write the method yourself:

```
class Page
  attr_accessor :color

  def color
    @color ||= 'blue'
  end
end
```

By doing this, we lose the concise one-line code which `attr_accessor` provides. We also lose some ability to visually scan the code and quickly see what's happening.

Fattr provides a much simpler way of defining default values:

```
require 'fattr'
class Page
  fattr :color => 'blue'
  fattrs :height => '768', :width => '1024'
end
```

The **fattr** methods generate the code we need to work like we have **attr_accessor** again. In this sample, Fattr uses heredocs to create a string to be used for the getter method:

```
code = <<-code
  def #{ name }(*value, &block)
    value.unshift block if block
    return self.send('#{ name }=', value.first) unless value.empty?
    #{ name }! unless defined? @#{ name }
    @#{ name }
  end
code
```

Once generated, the source of the method will look like this:

```
def color(*value, &block)
  value.unshift block if block
  return self.send('color=', value.first) unless value.empty?
  color! unless defined? @color
  @color
end
```

This **color** method generated by Fattr is equivalent to an **attr_reader** but it functions like an **attr_writer** if you provide it a value. Here's an example of how it works:

```
page = Page.new
page.color #=> 'blue'
page.color('red') #=> 'red'
page.color #=> 'red'
```

Fattr makes the decision that the getter method **color** will not merely be a getter, but can function as a declarative setter as well. By building your own DSL you can make decisions to provide this type of behavior or to restrict it.

Here's how the method works.

The call to `fattr` creates the methods `color`, `color=`, `color!`, and `color?`.

The current color is stored as an instance variable `@color` and will be returned from the `color` method:

```
def color(*value, &block)



  @color # return value
end
```

The stored value will be set using the `color!` method if `@color` has not already been defined:

```
def color(*value, &block)


  color! unless defined? @color
  @color # return value
end
```

The `color!` method will take the provided default value and set it to the related instance variable. In our case we specified `fattr :color => 'blue'` so calling `color!` will set `@color` to `'blue'`.

Because this is a declarative setter, it has to process any provided arguments.

Fattr allows declarative setters to use simple values or blocks, so each argument must handled independently. In particular, block arguments are favored over simple value arguments. The `*value` array of arguments will be altered to put any provide block first and it will use that to set the related instance variable:

```
def color(*value, &block)
  value.unshift block if block
  return self.send('color=', value.first) unless value.empty?
```

```
  color! unless defined? @color
  @color # return value
end
```

Ruby requires that block argumets appear last in a method signature.

Once the **value** array of arguments has the first element set to the block (to use it first), the code relies on the **color=** method to set the value. If neither a block nor other argument is passed, the code continues to either set the default, or return a previously set value.

Read through the short source code for Fattr for more ideas about its behavior and implementation.

Chapter 15

Example: FatoryGirl

A popular library for creating factories for objects is FactoryGirl.

The interface that FactoryGirl provides is simple and straightforward. Here's an example from it's "Getting Started" documentation:

```
FactoryGirl.define do
  # This will guess the User class
  factory :user do
    first_name "Kristoff"
    last_name  "Bjorgman"
    admin false
  end

  # This will use the User class (Admin would have been guessed)
  factory :admin, class: User do
    first_name "Admin"
    last_name  "User"
    admin      true
  end
end
```

A helpful DSL like this **`factory`** method will make good guesses and try to do the right thing, but it will allow you to override the default and configure the correct way to work.

Here, FactoryGirl uses **`instance_eval`** to execute the blocks of code passed to **`factory`**. But there's another interesting lesson inside of this library: callbacks.

FactoryGirl allows you to handle callbacks to perform some action, for example, after the object is created.

```
FactoryGirl.define do
  factory :user do
    first_name "Kristoff"
    last_name "Bjorgman"
    admin false

    after(:create) do |user, evaluator|
      create_list(:post, evaluator.posts_count, user: user)
    end
  end
end
```

In this sample, the **`after(:create)`** is run after the object is created, but the block accepts two arguments: **`user`** and **`evaluator`**. The **`user`** argument is the user that was created. The **`evaluator`** is an object which stores all the values created by the factory.

Let's take a look at how this is implemented:

```
def run(instance, evaluator)
  case block.arity
  when 1, -1 then syntax_runner.instance_exec(instance, &block)
  when 2 then syntax_runner.instance_exec(instance, evaluator, &block)
  else        syntax_runner.instance_exec(&block)
  end
end
```

FactoryGirl will create a callback object named by the argument given to the **`after`** method. The callback is created with the name **`:create`**, in this case, and with a block of code.

The block that we used in our example had two arguments.

The **`run`** method decides how to execute the code from the block.

The callback object stores the provided block and Ruby allows us to check the arity of the block, or in other words, it allows us to check the number of arguments.

When looking at a **`case`** statement, it's a good idea to check the **`else`** clause first. This gives you an idea of what will happen if there's no match for whatever code exists in the **`when`** parts.

There we see `syntax_runner.instance_exec(&block)` and this could easily be changed to use `instance_eval` instead. Ruby will evaluate, or execute, the block in the context of the `syntax_runner` object.

If the block's arity is greater than zero, FactoryGirl needs to provide the objects to the block so that our code works the way we expect.

The second part of the case checks if the block arity is equal to `2`.

```
when 2 then syntax_runner.instance_exec(instance, evaluator, &block)
```

If it is, the `syntax_runner` receives the `instance` (or in our case `user`) and the `evaluator`.

If, however, the arity is `1` or `-1` then the block will only receive the `instance` object.

So what is that `-1` value? Let's look at the ways we *could* create a callback:

```
# Two arguments and arity of 2
after(:create) do |user, evaluator|
  create_list(:post, evaluator.posts_count, user: user)
end
# One argument and arity of 1
after(:create) do |user|
  create_group(:people, user: user)
end
# Zero arguments and arity of 0
after(:create) do
  puts "Yay!"
end
# Any arguments and arity of -1
after(:create) do |*args|
  puts "The user is #{args.first}"
end
```

Ruby doesn't know how many `args` you'll give it with `*args` so it throws up it's hands and tells you that it's some strange number: `-1`.

This is the power of understanding how and when to use `instance_exec`; users of the DSL will expect it to make sense, and it will.

But wait! There's more!

What if you want to specify the same value for multiple attributes?

```
FactoryGirl.define do
  factory :user do
    first_name "Kristoff"
    last_name  "Bjorgman"

    password "12345"
    password_confirmation "12345"
  end
end
```

In the above example, both the **password** and **password_confirmation** are set to the same value. This could be bad. What if you change the password for one, but forget to change the other? If they are inherently tied in their implementation, then that could lead to some unexpected behavior when they are not the same.

I would, and probably you would too, prefer to tell FactoryGirl to just use the value I'd already configured:

Fortunately FactoryGirl allows us to use a great trick in Ruby using the **to_proc** method. Here's what it looks like in use:

```
FactoryGirl.define do
  factory :user do
    first_name "Kristoff"
    last_name  "Bjorgman"

    password "12345"
    password_confirmation &:password
  end
end
```

The important part is the **&:password** value provided to **password_confirmation**. Ruby will see the **&** character and treat the following as a block by calling **to_proc** on it. To implement this feature, FactoryGirl defines **to_proc** on attributes and there will use **instance_exec** to provide the symbol **password** to the block:

```
def to_proc
  block = @block
```

```
  -> {
    value = case block.arity
            when 1, -1 then instance_exec(self, &block)
            else instance_exec(&block)
            end
    raise SequenceAbuseError if FactoryGirl::Sequence === value
    value
  }
end
```

Chapter 16

Antipatterns

An antipattern is a common appoarch to a problem which produces a counter-productive result. Sometimes, however, there are valid reasons for doing things a certain way. It's best to study these examples and understand *why* they are counterproductive.

Metaprogramming can be tricky so it'll be good to avoid common pitfalls.

16.1 Defining methods on a class

We saw in the **has_many** example that methods were created on a module and include. The benefits of flexibility are clearly provide by our ability to use and override the initial implementation. If instead we define methods directly on the class, we'd paint ourselves into a corner.

Here's what our **has_many** could look like:

```
module HasMany
  def has_many(name)
    line_no = __LINE__; method_defs = %{
      def #{name}
        driver.find_elements(:css, '.#{name}')
      end

      def #{name}_texts
        #{name}.map(&:text)
      end
```

```
    }
    self.class_eval method_defs, __FILE__, line_no
  end
end

class Page
  extend HasMany

  has_many :features
end
```

This is extremely easy to implement. The downside is losing the ability to alter the behavior:

```
class Page
  extend HasMany

  has_many :features

  def features_texts
    super.map(&:strip)
  end
end
```

Upon using the `features_texts` method, Ruby will raise this error:

```
NoMethodError: super: no superclass method `features_texts' for #<Page:0x007fbff2ac9630>
```

There's no superclass because we defined the method directly on the class using `self.class_eval`. Avoid this problem by manipulating a module and including it in the inheritance hierarchy.

16.2 Switching scope

When creating a DSL, clarity and consistency should be high on our list of priorities.

When deciding between an implicit or explicit interface, it would be best to choose one and stick with it. As any programmer knows, the cost of context

switching can be immense. Likewise, *understanding* how parts of a DSL work can be difficult if the scoping of the blocks changes.

What would the Rails routing API feel like if it combined different approaches?

```
Rails.application.routes.draw do |map|
  map.resources :users
  map.resources :posts do
    resources :comments
  end
end

Rails.application.routes.draw do
  resources :users
  resources :posts do |posts|
    posts.resources :comments
  end
end
```

Neither of the above options feel quite right. In the first example there would be a jarring context switch between using `map.resources` and the bare `resources` in the block. The second example at least feels more comfortable but could lead to confusion in the future: when does `resources` need an explicit receiver and when does it not?

You may decide to use both approaches in different parts of your program, but switching between them will likely be a costly mental change for the users of your DSL. In some cases, you may find that this mental stumbling block can be used as a warning sign that part of the DSL must be treated differently. If you do create a syntax which requires that change in semantics, be careful.

16.3 Automatic scope switch

This pattern can feel useful and powerful but can end up being confusing for developers getting to know your DSL.

You can create an implementation which allows you to provide block arguments or not. Here's an example:

```
process do
  start
end

process do |page|
  page.start
end
```

We're looking at code which switches automatically between implicit and explicit interfaces for the object of concern. Here's how you could implement it:

```
def process(&block)
  if block.arity.zero?
    self.instance_eval(&block)
  else
    block.call(self)
  end
end
```

The **`arity`** of a block represents the number of arguments it accepts.

If you create a block with **`process do`** it will have an arity of zero (0). If you create a block with **`process do |page|`** it will have an arity of one (1).

While the conceptual difference between one argument or zero is simple and clear, the difference between how they are executed is significant.

This approach feels like a nice "catch all" for handling blocks of code, but it is so flexible that it's usefulness could be hidden by it's undecided use cases.

Think carefully before introducing a flexible but potentially confusing interface to your code.

Chapter 17

Additional Resources

This book can only scratch the surface of what's available to build DSLs.

17.1 Docile

Docile is a project designed to help making DSLs easy and provide tools to overcome the limitations around choosing and implicit or explicit interface.

In the example we wrote, we saw a failure by attempting to access a local variable created outside of the scope of the block:

```
marshmallow = OpenStruct.new(angry: true)

Olaf.configure do
  unless marshmallow.angry
    likes_warm_hugs
  end
end

# => NoMethodError: undefined method `angry' for nil:NilClass
```

With Docile, we could introduce access to instance variables to overcome this restriction:

```
module Olaf
  def self.configure(&block)
    # instance_eval(&block) # <--- Previous implementation
    Docile.dsl_eval(self, &block) # <--- access instance variables from outer scope
  end
end

@marshmallow = OpenStruct.new(angry: true)

Olaf.configure do
  unless @marshmallow.angry
    likes_warm_hugs
  end
end
```

Explore the Docile code for more ideas about how to manage scope and implementation constraints.

17.2 Rebuil

Rebuil is designed to be a "readable DSL for Ruby Regexp." This is a project designed to simplify the built-in DSL for Regular Expression processing.

17.3 Verbal

Verbal is based on a JavaScript library designed to simplify Regular Expressions.

17.4 RLTK

RLTK is the "Ruby Language Toolkit" and can be used to build support for external DSLs.

Tempo is a web template system built with RLTK.