Extracted from:

The ThoughtWorks Anthology

Essays on Software Technology and Innovation

Chapter 3

One Lair and Twenty Ruby DSLs

by Martin Fowler, Chief Scientist

Much of the reason for Ruby's recent popularity is its suitability as a base for writing internal domain-specific languages. Internal DSLs are domain-specific languages written using a valid subset of a host language. There's a resurgence about doing them in the Ruby at the moment.

Internal DSLs are an old idea particularly popular in Lisp circles. Many Lispers dismiss Ruby as having nothing new to offer in this space. One feature that does make Ruby interesting is the wide range of different techniques you can use in the language to develop an internal DSL. Lisp gives you some great mechanisms but relatively few compared to Ruby, which offers many options.

My purpose in this essay is to explore lots of these options for a single example so you have a sense of the possibilities and so you can consider which techniques work for you more than others.

3.1 My Lair Example

For the rest of this chapter I'll use a simple example to explore the alternative techniques. The example is a common, interesting abstract problem of configuration. You see this in all sorts of equipment: if you want *x*, you need to have a compatible *y*. You see this configuration problem when buying computers, installing software, and doing lots of other less nerdy pursuits.

For this particular case, imagine a company that specializes in providing complex equipment to evil megalomaniacs who want to conquer the world. Judging by the amount of films about them, it's a large market—and one made better by the fact that these lairs keeping getting blown up by glamorous secret agents.

So, my DSL will express the configuration rules for things that megalomaniacs put in lairs. This example DSL will involve two kinds of things: items and resources. Items are concrete things such as cameras and acid baths. Resources are amounts of stuff you need, like electricity.

I have two kinds of resources in my example: electricity and acid. I assume that resources have potentially lots of different properties that need to be matched. For instance, I'll need to check that all the items' power needs are supplied by power plants in the lair (evil geniuses don't like bothering with utilities). As a result, each resource will be implemented by its own class in my abstract representation.

For the sake of the problem, I assume resources fall into two categories, simple ones that have a small, fixed number of properties that can thus be rendered as arguments in the constructor (electricity) and complex ones with many optional properties that need lots of setting methods (acid). Acid actually has only two properties for this example, but just imagine there are dozens of them.

When it comes to items, I can say three things about them: they use resources, they provide resources, and they depend on another item that needs to be present in the lair.

Now for the curious, here's the implementation of this abstract representation. I'll use the same abstract representation for all the examples I'll discuss:

Download lairs/model.rb

```
class Item
  attr_reader :id, :uses, :provisions, :dependencies
  def initialize id
    @id = id
    @uses = []
    @provisions = []
    @dependencies = []
  end
  def add_usage anItem
    @uses << anItem
  end
```

```
  def add_provision anItem
    @provisions << anItem
  end
  def add_dependency anItem
    @dependencies << anItem
  end
end

class Acid
  attr_accessor :type, :grade
end

class Electricity
  def initialize power
    @power = power
  end
  attr_reader :power
end
```

I store any particular configuration in a configuration object:

Download lairs/model.rb

```
class Configuration
  def initialize
    @items = {}
  end
  def add_item arg
    @items[arg.id] = arg
  end
  def [] arg
    return @items[arg]
  end
  def items
    @items.values
  end
end
```

For the purpose of this chapter, I'll define just a few items and their rules:

- An acid bath uses 12 units of electricity and grade-5 hydrochloric acid (HCl).
- A camera uses 1 unit of electricity.
- A small power plant provides 11 units of electricity and depends on a secure air vent in the lair.

I can state these rules in terms of the abstract representation like this:

Download lairs/rules0.rb

```
config = Configuration.new
config.add_item(Item.new(:secure_air_vent))

config.add_item(Item.new(:acid_bath))
config[:acid_bath].add_usage(Electricity.new(12))
acid = Acid.new
config[:acid_bath].add_usage(acid)
acid.type = :hcl
acid.grade = 5

config.add_item(Item.new(:camera))
config[:camera].add_usage(Electricity.new(1))

config.add_item(Item.new(:small_power_plant))
config[:small_power_plant].add_provision(Electricity.new(11))
config[:small_power_plant].add_dependency(config[:secure_air_vent])
```

Although this code populates the configuration, it isn't very fluent. The rest of this chapter explores different ways of writing code to express these rules in a better way.

3.2 Using Global Functions

Functions are the most basic structuring mechanism in programming. They provide the earliest way to structure software and to introduce domain names into a program.

So, my first attempt at a DSL might be to use a sequence of global function calls:

Download lairs/rules8.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(acid)
acid_type(:hcl)
acid_grade(5)
uses(electricity(12))

item(:camera)
uses(electricity(1))

item(:small_power_plant)
provides(electricity(11))
depends(:secure_air_vent)
```

The function names introduce the vocabulary of the DSL: **item** declares an item, and **uses** indicates that an item uses a resource.

The configuration rules in this DSL are all about relationships. When I say a camera uses 1 unit of electricity, I want to make a link between an item called *camera* and an electricity resource. In this first lair expression, this linkage is done through a context established by the sequence of commands. The line uses(electricity(1)) applies to the camera item because it immediately follows the declaration of *camera*. I might say that this relationship is defined implicitly by the *sequential context* of the statements.

As a human, you can infer the sequential context by how you read the DSL text. When processing the DSL, however, the computer needs a bit more help. To keep track of the context, I use special variables as I load the DSL; unsurprisingly, they're called *context variables*. One context variable keeps track of the current item:

Download lairs/builder8.rb

```
def item name
  $current_item = Item.new(name)
  $config.add_item $current_item
end

def uses resource
  $current_item.add_usage(resource)
end
```

Since I am using global functions, I need to use global variables for my context variables. This isn't that great, but as you'll see, there are ways to avoid this in many languages. Indeed, using global functions is hardly ideal either, but it serves as a starting point.

I can use the same trick to handle the properties of the acid:

Download lairs/builder8.rb

```
def acid
  $current_acid = Acid.new
end
def acid_type type
  $current_acid.type = type
end
```

Sequential context works for the links between an item and its resources but is not very good for handling the nonhierarchical links between dependent items. Here I need to make explicit relationships between items. I can do this by giving an item an *identifier* when I

declare it (item(:secure_air_vent)) and using that identifier when I need to refer to it later (depends(:secure_air_vent). The fact that it is the small power plant that depends on the secure air vent is handled through sequential context.

A useful distinction here is that the resources are what Evans calls value objects [Eva03]. As a result, they aren't referred to other than by their owning item. Items themselves, however, can be referred to in any way in the DSL through the dependency relationship. As a result, items need some kind of identifier so that I can refer to them later.

The Ruby way of handling an identifier like this is to use a symbol data type: :secure_air_vent. A symbol in Ruby is a sequence of nonwhitespace characters beginning with a colon. Symbol data types aren't in many mainstream languages. You can think of them as like strings, but for the particular purpose of this kind of usage. As a result, you can't do many of the usual string operations on them, and they are also designed so all uses of them share the same instance. This makes them more efficient for lookups. However, I find the most important reason to use them is that they indicate my intent of how I treat them. I'm using :secure_air_vent as a symbol, not a string, so picking the right data type makes my intent clear.

Another way of doing this, of course, is to use variables. I tend to shy away from variables in a DSL. The problem with variables is that they are variable. The fact that I can put a different object in the same variable means I have to keep track of which object is in which variable. Variables are a useful facility, but they are awkward to keep track of. For DSLs I can usually avoid them. The difference between an identifier and a variable is that an identifier will always refer to the same object—it doesn't vary.

Identifiers are necessary for the dependency relationship, but they can also be used to handle resources as an alternative to using sequential context:

Download lairs/rules7.rb

```
item(:secure_air_vent)

item(:acid_bath)
uses(:acid_bath, acid(:acid_bath_acid))
acid_type(:acid_bath_acid, :hcl)
acid_grade(:acid_bath_acid, 5)
uses(:acid_bath, electricity(12))
```

```
item(:camera)
uses(:camera, electricity(1))

item(:small_power_plant)
provides(:small_power_plant, electricity(11))
depends(:small_power_plant, :secure_air_vent)
```

Using identifiers like this means I'm being explicit about the relationships, and it also allows me to avoid using global context variables. These are both usually good things: I do like being explicit, and I don't like global variables. However, the cost here is a much more verbose DSL. I think it's valuable to use some form of implicit mechanism in order to make the DSL more readable.

3.3 Using Objects

One of the principal problems of using functions as I did earlier is that I have to define global functions for the language. A large set of global functions can be difficult to manage. One of the advantages of using objects is that I can organize my functions by classes. By arranging my DSL code properly, I can keep the DSL functions collected together and out of any global function space.

Class Methods and Method Chaining

The most obvious way to control the scope of methods in an object-oriented language is to use class methods. Class methods do help scope the use of functions but also introduce repetition because the class name has to be used with each call. I can reduce the amount of that repetition considerably by pairing the class methods with method chaining, as in this example:

Download lairs/rules11.rb

```
Configuration.item(:secure_air_vent)

Configuration.item(:acid_bath).
    uses(Resources.acid.
      set_type(:hcl).
      set_grade(5)).
    uses(Resources.electricity(12))

Configuration.item(:camera).uses(Resources.electricity(1))

Configuration.item(:small_power_plant).
    provides(Resources.electricity(11)).
    depends_on(:secure_air_vent)
```

Here I begin each of my DSL clauses with a call to a class method. That class method returns an object that is used as a receiver for the next call. I can then repeatedly return the object for the next call to chain together multiple method calls. In some places, the method chaining becomes a little awkward, so I use class methods again.

It's worth digging into this example in more detail so you can see what's happening. As you do this, remember that this example does have some faults that I'll explore, and remedy, in some later examples.

I'll begin with the opening of the definition of an item:

Download lairs/builder11.rb

```
def self.item arg
  new_item = Item.new(arg)
  @@current.add_item new_item
  return new_item
end
```

This method creates a new item, puts it into a configuration stored in a class variable, and returns it. Returning the newly created item is the key here, because this sets up the method chain.

Download lairs/builder11.rb

```
def provides arg
  add_provision arg
  return self
end
```

The provides method just calls the regular adder but again returns itself. This continues the chain, and the other methods work the same way.

Using method chaining like this is at odds with a lot of good programming advice. In many languages the convention is that modifiers (methods that change an object's state) do not return anything. This follows the principle of command query separation, which is a good and useful principle and one that's worth following most of the time. Unfortunately, it is at odds with a flowing internal DSL. As a result, DSL writers usually decide to drop this principle while they are within DSL code in order to support method chaining. This example also uses method chaining to set the type and grade of acid.

A further change from regular code guidelines is a different approach to formatting. In this case, I've laid out the code to emphasize the hierar-

chy that the DSL suggests. With method chaining you often see method calls broken over newlines.

As well as demonstrating method chaining, this example demonstrates how to use a factory class to create resources. Rather than add methods to the Electricity class, I define a resources class that contains class methods to create instances of electricity and acid. Such factories are often called *class factories* or *static factories* because they contain only class (static) methods for creating appropriate objects. They can often make DSLs more readable, and you avoid putting extra methods on the actual model classes.

This highlights one of the problems with this DSL fragment. To make this work, I have to add a number of methods to the domain classes—methods that don't sit well. Most methods on an object should make sense as individual calls. But DSL methods are written to make sense within the context of DSL expressions. As a result, the naming, as well as principles such as command query separation, are different. Furthermore, DSL methods are very context specific, and they should be used only within DSL expressions when creating objects. Basically, the principles for good DSL methods aren't the same as what makes regular methods work effectively.

Expression Builder

A way of avoiding these clashes between DSLs and regular APIs is to use the Expression Builder pattern. Essentially this says that the methods that are used in a DSL should be defined on a separate object that creates the real domain object. You can use the Expression Builder pattern in a couple of ways. One route here is to use the same DSL language but to create builder objects instead of domain objects.

To do this, I can change my initial class method call to return a different item builder object:

Download lairs/builder12.rb

```
def self.item arg
  new_item = ItemBuilder.new(arg)
  @@current.add_item new_item.subject
  return new_item
end
```

The item builder supports the DSL methods and translates these onto methods on the real item object.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

ThoughtWorks Anthology's Home Page
http://pragprog.com/titles/twa
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/twa.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com