

Jeremy McAnally  
Assaf Arkin

# RUBY

IN PRACTICE



MEAP

Unedited Draft

 MANNING



**MEAP Edition  
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=356>

## **Table of Contents**

- 1. *Ruby under Microscope***
  - 2. *Testing Ruby***
  - 3. *Integration with Ruby***
  - 4. *Rails Techniques***
  - 5. *Web Services***
  - 6. *Automating Communication***
  - 7. *Asynchronous Messaging***
  - 8. *Deployment and Management***
  - 9. *Databases***
  - 10. *Structured Documents***
  - 11. *Authentication and Identity Brokering***
  - 12. *Indexing and Searching***
  - 13. *Document Processing and Reporting***
- Appendix A: Improving Ruby performance***
- Appendix B: Bridging Ruby (e.g., JRuby, RubyCLR, SWIG, etc.)***
- Appendix C: Your Development Environment***

# *Ruby under the Microscope*

Often people, especially computer engineers, focus on the machines. They think, “By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something.” They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.

## **YUKIHIRO MATSUMOTO, CREATOR OF RUBY**

You’ve heard it all before, right? A new language or framework becomes the flavor du jour and everyone starts talking about it. First there’s a low rumble on websites, then someone gets a hold of it, does something cool, and out comes the marketing speak. I’m sure you can imagine Dave from marketing barking at you about another amazing technology: “You will be more productive! Our synergistic approach to dynamic, domain-driven, development will allow you to get to market quicker and get a better return on investment! Get a lower TCO and higher ROI over J2EE with our XP driven Scrum model based on XML! Take apart your FOB and overhaul your BOB with our easy to use, turnkey solution!” To some in the world of software development, it sounds like Ruby is experiencing hype and buzz, but we’re here and this book is meant to show you that you really can develop “real” software with Ruby.

Maybe you heard the accolades and decided to read this book to find out if Ruby is right for you. Maybe you know Ruby already and you chose this book to pick up some practical techniques you can take back to the workplace.

## **1.1 Why Ruby now?**

Here’s a little known fact that surprises many people: Ruby came to the world in 1995, the same year as Java. Like many other open source technologies (like Linux and MySQL) it took its time to mature and get noticed. So what happened in those ten years that turned Ruby from a little known language into a hot item ticket without the help of a big vendor marketing machine? The adoption of Ruby on Rails, Ruby’s premier Web development framework, is the obvious answer, and it has without a doubt skyrocketed Ruby’s popularity. It brought on hordes of developers who use Ruby exclusively with Rails, and even more developers who came for Rails, but stayed for the Ruby. But while Rails played a major role in getting Ruby into the mainstream, it still doesn’t explain why it happened only recently, and not before.

### **1.1.1 Dealing with complexity**

If you work for a big company, chances are you have to deal with complex problems. Sales across different channels, multiple products and markets, suppliers and distributors, employees and contractors, accounting and SOX compliance, market dynamics and regulations, and the list goes on and on. It’s unavoidable: the problems of running any sizable business are complex. But what about

the solutions? You're probably thinking, there are no simple solutions to complex problems, and complexity is the nature of any real business. But do solutions have to be unnecessarily complex?

That's true, but given the complexity that naturally arises from these business problems you are trying to solve, you don't want the technology you need to solve them to be unnecessarily complex. Even further, the more technology you throw at the problem – Web servers and databases, on-line and batch processing, messaging protocols and data formats – the more complexity you add. The only way to alleviate this complexity conundrum is by looking for simpler solutions to existing problems.

## ***Optimizing developer cycles***

In recent years there has been a growing realization that software companies are targeting these business problems (the “enterprise” space) by offering overly complex solutions. We can't blame them. Complex solutions sell better. It's easier to obfuscate the solution and abstract the problem to enchant executives into buying your software than to design a solid, simple solution. As more developers are realizing that these “silver bullets” solutions create more problems than they solve, the momentum is shifting towards simpler, lightweight technologies.

### **OPEN SOURCE**

It seems that open source, with its organic development model, is able to adapt to this changing of tides better. For example, in the Java space, once can see a strong bias towards Spring and Hibernate as opposed to EJB. Many developers are defecting from a lot of spaces to Rails. Why? Because those projects aren't afraid to re-evaluate their approaches to accommodate current developer attitudes, especially since these sorts of projects are developed by the developers who use them everyday in their own work.

We like to talk about large scale systems, thousands of servers, petabytes of data, billions of requests. It's captivating, the same way we could talk about horsepower and 0 to 60 acceleration times. But in real life we often face constraints of a different scale. Can you do it with a smaller team? Can you get it done tomorrow? Can you add these new features before we go into beta? Most often in real situations businesses have to optimize not for CPU cycles, but developer cycles. It's easy to scale out by throwing more hardware at the problem, but as many businesses have found out, throwing more people at the problem just makes the project late. Sure that knowledge was captured years ago in Fred Brooks' *Mythical Man Month*, but our bosses just decided to prove it empirically.

Minimizing developer cycles is probably the single most attractive feature for dynamic languages, and Ruby in particular. Simplifying software development has been the holy grail of the software industry. Say what you will about COBOL, it's much better than writing mainframe applications in assembly language. And believe it or not, productivity was a major selling point for Java, in the early days when it came to replace C/C++. It's the nature of the software development that every once in a while we take a leap forward by changing the way we write code to deal with the growing complexity that came since the last major leap. And it's not the sole domain of the language and its syntax. One of the biggest criticisms against J2EE is the sheer size of its API, and the complexity involve in writing even the simplest of programs. EJB is the poster child of developer unfriendly technology.

The true measure of a programming language's productivity is how little code you need in order to solve a given problem. Writing less code while being able to do the same thing will make you far more productive than writing a whole lot of code without doing much at all. This is a reverse relationship compared to how many businesses view productivity: lines of code produced. Of course,

if simple lines of code were the metric, Perl would win every time. Just like too much code can make your application unmaintainable, so can terse, short code that's "write only."

## **Language features**

Ruby seems to hit the soft spot and appeal to developers that value natural interfaces and have a strong contingency towards migrating away from languages that inherently promote complexity. But why? Why would developers move away from "proven" technologies to Ruby, arguably, the "new kid," regardless of its positive aspects? Primarily, it's because Ruby is a dynamic language that works well for applications and scripting, supports the object-oriented and functional programming styles, bakes arrays and hash literals into the syntax, with just enough meta-programming features to allow domain-specific languages. Had enough marketing? Of course, this laundry list of buzzwords is not as important as what happens when you combine all these features together. They go from buzzwords and abstract concepts to become a powerful tool.

For example, consider Ruby On Rails. Rails is one incarnation in a long heritage of Web application frameworks. Like so many Web application frameworks before it, Rails deals with user interface and remote APIs, business logic, and persistence. Unlike so many Web application frameworks before it, it does so effortlessly without taxing the developer. In its three years of existence it leapfrogged the more established frameworks to become the benchmark by which all other frameworks are judged.

All that power comes from Ruby. The simplicity of mapping relational databases to objects without the burden of XML configuration is a combination of object-oriented and dynamic styles. The ease of writing HTML and XML templates, and setting up filters comes from functional programming. Magic features like dynamic finders and friendly URL routing are all forms of meta-programming. The little configuration it needs is handled effortlessly using a set of domain-specific languages. It's not that Rails (or really Ruby) is doing anything new, but the attractiveness comes from *how* it does things. Besides being a successful framework on its own, Rails showed the world how to use Ruby's combination of language features to create applications that, quite frankly, rock. Dynamic features like `method_missing` and closures go beyond conceptual curiosity and help you deliver.

So, why is Ruby popular now? It can for the most part be traced to developers growing weary of complex, taxing development tools and the emergence of Rails as a definite, tangible project that shows that Ruby can be used to create production quality software that's still developer friendly. Once you start working with Ruby, you'll probably realize this, too. It's about the same amount of effort to work with flat files, produce PDFs, make SOAP requests, send messages using WebSphere MQ as it is to map objects to databases, send e-mail, or parse an XML file. Wrap that into a nice, natural syntax, and you have a very potent tool for software development.

## **1.2 Ruby by Example**

We think the best way to illustrate this idea is by example, and what better way than diving into some code? Let's take a fairly typical situation and do something with it: you need to pull some data from a database and create some graphs with it. Perhaps you need to trace the sales performance of a range of products over across all of your sales locations. Can we keep it simple?

First, we'll need to install three libraries: ActiveRecord (an object-relational mapper for databases), Scruffy (a graphing solution for Ruby), and Rmagick (ImageMagick bindings for Ruby, required by Scruffy). Let's do that using the RubyGems utility:

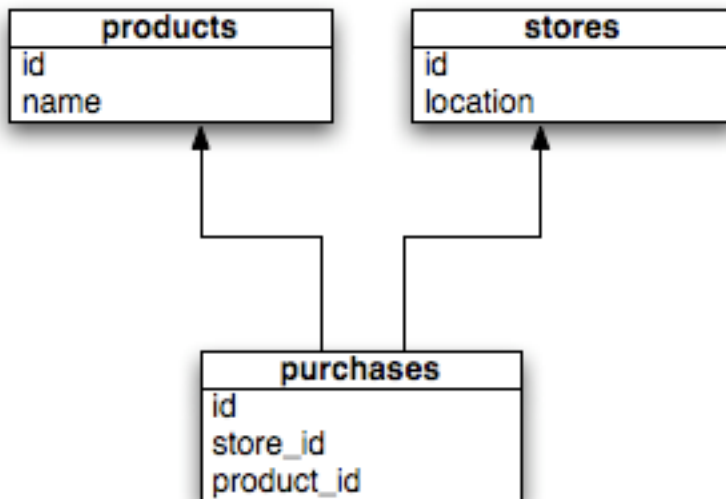
```
gem install active_record
gem install rmagick
gem install scruffy
```

Assuming you have all the system prerequisites for these packages (for example, ImageMagick for RMagick to bind to), you should now have all you need.

### **RMAGICK**

RMagick can be a beast to setup. We suggest checking the project's documentation, the mailing list, and your favorite search engine if you are having problems.

Now, let's setup our database; Figure 1.1 below shows our schema diagram for the database. You can use that as a model to create the tables (if you prefer to use some sort of GUI tool), or you can use the included SQL below it.



**Figure 1.1** For our graph, we will build a simple domain model: Products will have purchases which belong to the stores where they happened.

**Listing 1.1 SQL for graph example database**

```
CREATE DATABASE `paper`;  
  
CREATE TABLE `products` (  
  `id` int(50) NOT NULL auto_increment,  
  `name` text,  
  PRIMARY KEY (`id`)  
);  
  
CREATE TABLE `purchases` (  
  `id` int(50) NOT NULL auto_increment,  
  `product_id` int(50) default NULL,  
  `store_id` int(50) default NULL,  
  PRIMARY KEY (`id`)  
);  
  
CREATE TABLE `stores` (  
  `id` int(50) NOT NULL auto_increment,  
  `location` text,  
  PRIMARY KEY (`id`)  
);
```

Now, let's set up ActiveRecord to work with the database. ActiveRecord is typically used inside of a Rails application, and since we're not using it in that environment it takes a few more lines of configuration.

**Listing 1.2 Setting up our database with ActiveRecord**

```
require `rubygems' #1  
require_gem `activerecord' |  
  
# Begin ActiveRecord configuration  
ActiveRecord::Base.establish_connection( #2  
  :adapter => "mysql", |  
  :host => "localhost", |  
  :database => "paper", |  
  :username => "root", |  
  :password => "" |  
) |  
  
# Begin ActiveRecord classes  
class Product < ActiveRecord::Base #3  
  has_many :purchases  
end  
  
class Purchase < ActiveRecord::Base  
  belongs_to :store  
end  
  
class Store < ActiveRecord::Base  
end  
  
# Begin logic
```

```

myproducts = Product.find(:all) #4

require 'pp' #5
pp myproducts |
<#1 Require RubyGems>
<#2 Configure ActiveRecord>
<#3 Build our ActiveRecord classes>
<#4 Search for all products>
<#5 Use Pretty Printing to output our collection>

```

As you can see, it doesn't take a lot of code to get a full object-rationally mapped database connection. First, you need to import the RubyGems library (#1) so you can then import ActiveRecord. Next, you need to establish a database connection with ActiveRecord. Normally this configuration data would live in a database configuration file in a Rails application (i.e., `database.yml`), but for this example we chose to run outside Rails, so we'll use `establish_connection` directly. Next we need to create ActiveRecord classes and associations to map our database #3.

### ACTIVERECORD OFF RAILS

We discuss using ActiveRecord without Rails a little later on in Chapter 6, Databases.

Finally, we execute a query (#4) and output its results using Pretty Printing (`pp`). Now you just need to fill in some testing data (or download the script from the book's source code to generate some for you), and run the script. You should see something like the following output.

```

[#<Product:0x639e30 @attributes={"name"=>"Envelopes", "id"=>"1"}>,
 #<Product:0x639e08 @attributes={"name"=>"Paper", "id"=>"2"}>,
 #<Product:0x639c00 @attributes={"name"=>"Folders", "id"=>"3"}>,
 #<Product:0x639bb0 @attributes={"name"=>"Cardstock", "id"=>"4"}>]

```

Alright, so our database is set up and our query works, so let's move on to generating a graph from the data. First, remove those last two lines from the previous listing if you'd like (they will be superfluous by the time we're done). Let's take that data that we retrieved, process it, and build the graph using Scruffy; in listing 1.3, you'll see how to do that.

#### Listing 1.3 Generating the graph with Scruffy

```

graph = Scruffy::Graph.new #1
graph.title = "Comparison of Product Sales" |

stores = Store.find(:all, :select=>"id, location")

myproducts.each do |product| #2
  counts = stores.collect do |store|
    Purchase.count "id", :store_id=>store.id, :product_id=>product.id
  end

  graph.add :line, product.name, counts #3
end

graph.point_markers = #4
  Store.find(:all).map! {|store| store.location} |
graph.render :as => 'PNG', :to => 'productsales.png',

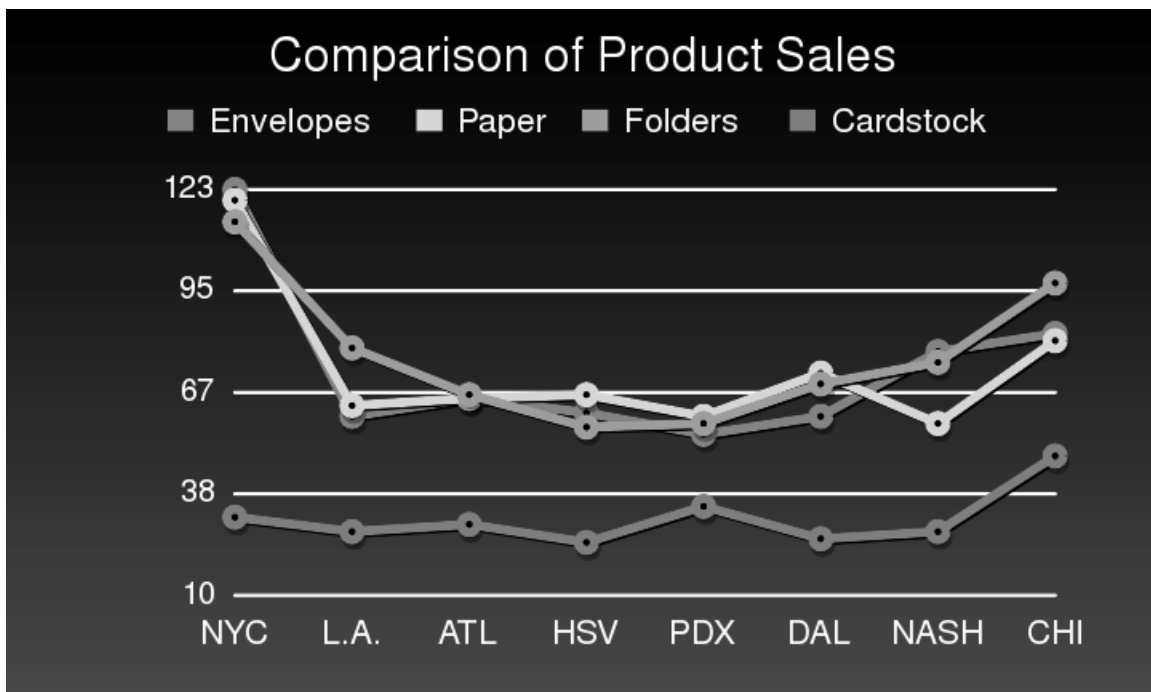
```

```

        :width => 720,
        :theme => Scruffy::Themes::Keynote.new)      #5
<#1 Create a new Scruffy object and give it a title>
<#2 Iterate the products>
<#3 Add a line to the graph for each>
<#4 Put in point markers>
<#5 Generate the graph with a theme>

```

First, we need to create a `Scruffy::Graph` instance and do a little bit of setup #1. Next, we iterate the products we found earlier and calculate the sales counts for each store location #2. Finally, we add a line on the graph showing the sales trends of that product across the stores #3. Before we render the graph, we need to add the markers to indicate which sales location we're looking at #4; finally, we render the graph to a PNG file using one of Scruffy's built-in themes #5. If you open the graph up and look at it, you can see that we have a very polished graph (ours looks like Figure 1.2).



**Figure 1.2** Our finished product. In about 40 lines of code we pulled data from the database, processed it, and graphed it in a rather attractive fashion.

Not bad for 40 lines of code, including whitespace, comments, and more verbose than required constructs. Sure this example isn't representative of every situation (you can't develop a full CRM solution in Ruby with 40 lines of code!), but it speaks volumes about the expressiveness of the language and the power of its toolkit. In the next section and chapters, we'll take a look at a lot of the concepts that powered this example, so you can start building applications and tools that take full advantage of Ruby's features.

## 1.3 Facets of Ruby

Now that we've spent some time discussing the "why" of Ruby, we'll be spending some time looking at the "how." One of the goals of this book is to make you into a truly effective Ruby developer; we want you to become Ruby crafts(wo)men, able to use Ruby to reframe problems and

craft solutions. In this section, we'll discuss some Ruby concepts and unique "Rubyisms" that will help you do this and power the examples and libraries you'll see throughout the rest of the book. Our goal is that you'll come away with a grasp of these advanced concepts and how to craft code that is readable, expressive, and "good" (whatever subjective method you use to measure that). If we're successful, you'll be able to use Ruby to do your job more effectively and develop more maintainable applications.

But what do we mean when we say "reframing in Ruby"? Every programming language has its own set of idioms and best practices. Once you're comfortable with the syntax, and know your way around the libraries, you start to explore that which makes the language unique. You explore the character of the language, if you will: the way it promotes a certain style of programming and rewards you for following it. Work with the language and it will work for you.

Object oriented languages, for example, ask you to encapsulate behavior and data into objects, and reward you for that in reuse. If you're coming from Java, you know the value of using JavaBeans, the standard collections library, throwing and catching exceptions, and so on. Today we take those for granted, but in the early days of Java development, many developers would use Java as if it was C or Visual Basic. They wrote code that didn't follow Java idioms, which made it harder to maintain and use.

Like Java, Ruby has its own set of idioms. For example, in Ruby you use blocks often to keep your code simple and readable. You can write methods that extend classes with new functionality (i.e., meta-programming). You enrich classes with common behavior by mixing in modules, fondly known as *mixins*. You can use *blocks* to abstract loops and even extend methods and reduce code duplication. For example, let's say you were writing a script to interface with an old legacy server; it accepts TCP connections and operates on very simple commands like LOGIN, GET, DELETE, and so on. Each time you start a session with the server, you need the same setup and teardown, but you want to do different things during the socket's connection each time. You could write a number of methods for each sequence of events, duplicating the setup and teardown code in each one, or you could use a block; listing 1.4 shows a simple implementation of this script.

**Listing 1.4 Using blocks to reduce code duplication.**

```
def remote_session
  config = get_configuration_information
  sock = TCPSocket::new(config.host, config.port)
  yield sock #1
  sock.close
  log_results
end

# Simply send some data
def send_login #2
  remote_session {|sock| sock.send('LOGIN')}
end

# A little more advanced
def login #3
  remote_session do |sock| #4
    sock.send('LOGIN')
    received = sock.recv(128)
    if received == 'OK'
      true
    end
  end
end
```

```
    else
      false
    end
  end
end
end
```

|

First, we create a method to execute our setup and teardown, with a `yield` statement inside #1. That `yield` statement tells Ruby to execute the block that is fed to the method as a parameter. Next, we create a simple method to send the `LOGIN` command to our server (#2); note that a block is fed to the method as a parameter. The `sock` parameter is the socket from our “setup” method (`remote_session`) that is given to the block to use. We do the same for the `login` method (#3), except this time we send the `login` and do something with the returned data (#4). Notice the code duplication we’ve eliminated by putting all of our setup and teardown into a separate method; this is just one of the many facets of Ruby that make development with it that much cleaner and easier.

Ruby is a dynamic language, and as you get to explore that facet of Ruby, known as *duck typing*, you’ll notice that you don’t need to use interfaces and abstract classes that often. The way Ruby allows you to extend objects and use constructors, relieves you from working juggling factories. Iteration is often done with blocks and functions, whereas in non-functional languages you tend to use *for* loops and define anonymous classes.

Once you get a hang of all of these ideas, you start reframing problems in terms of Ruby, rather than thinking out a solution in a language you’re more familiar with, and then writing it in the Ruby syntax. It will change your thought process from problem to Java to Ruby, or problem to C++ to Ruby to working directly with Ruby. You will often find that reframing problems changes the way you think about code, and you realize new ways to become a better developer. For example, imagine a method that queries the database for products based on different criteria. You can call the method with any number of arguments. If your code looks like `find_products nil, nil, nil, nil, 5, 50, 250`, it’s hard to understand exactly what criteria it’s using. What does the 5 stand for? What about the 50?

Some languages promote the idiom of method overloading, creating several methods with different signatures, depending on the expected call. Overloading doesn’t always work, for example, if all the arguments are integers, you’ll want to follow a different idiom, by creating an object, populating its fields and then passing it to the method. Those are all good idioms, but not the one you’ll likely to use with Ruby.

Ruby has a convenient syntax for handling hashes, which you can use to simulate named arguments. Instead of method overloading and populating objects, you can write something like `find_products :category=>5, :price=>50..250`. Duck typing even makes it possible to call `find_products :category=>storage, :price=>50..250` and let the method extract the identifier from the `storage` object. As you practice these idioms, you’ll notice certain things change. For one, you’ll have fewer methods in your objects, making your APIs easier to understand. In fact, you’ll have fewer classes to deal with, in this example we’ve eliminated the need for an object to hold all the properties and a class to define it. With less classes and methods, understanding the overall design of your software and how all the pieces fit together becomes easier.

Our discussion has barely scratched the surface. This book is not an introduction to Ruby, we only wanted to give you a taste for the language. We could go on and on about all of the dynamic features of Ruby, but it would ultimately be redundant. We’ll cover some of these concepts, features, and practices in this chapter directly, but throughout the book you’ll get to see more Ruby idioms in practice. We hope that through example and explanation, we can help you learn those by example.

The more you invest in learning the language, in getting under the hood and really getting your hands dirty with Ruby, the better developer you'll be. There's always a learning curve, but fortunately with Ruby, there's not a lot to it. You'll see the benefit of Ruby as you're learning and practicing the language, but you'll want to take that information and go deeper to get even better.

One of the killer apps for Ruby is the Ruby on Rails Web application framework. Developers are very attracted to the magic of Rails the productivity gains therein, but, in reality, most of the "magic" is really just good Ruby programming applied to Web applications. Think about this: what if you could make the same magic work for you in other domains, to make you a better developer in any problem you need to solve? What if you could take those same productivity gains and "magic" methods and use them in *your* code? Let's take a look at some of the facets of this gem we call Ruby that can make this possible: duck typing, simplicity, efficiency, and functional programming.

### 1.3.1 Duck typing

Ruby uses dynamic typing rather than static typing. In a static typing system, an objects' type is determined by its class definition at compile time. Static typing forces each object to declare its heritage, you're always asking, "Where are you coming from?" In dynamic languages behavior is captured by the object not the interface. Dynamic typing only cares about merits, the question to ask each object is, "What can you do?"

You can do that same with reflection in Java or C#, but reflection hides your business logic in a haystack of type-bypassing code. With dynamic typing you don't have to declare so many interfaces and abstract classes in anticipation of reuse; you don't have to write adapters and decorators as often; you don't need to choose between readability and functionality. All these help you reuse code more often.

A by-product of duck typing is that method calls do not check their recipient's type beforehand. Your code calls a method, and if it fails, raises an exception. This concept sounds a little cloudy, so let's look at a piece of code to explain it. Let's say we have a method that calls *size* and returns it in a friendly message.

```
def print_size(item)
  puts "The item's size is #{item.size}."
end
```

Our method simply calls *size* on the object without regard for its class; so if you feed it an object that responds to the *size* method, it will work.

```
mystring = "This is a string."
print_size(mystring)           # => The item's size is 17.
myarray = [1,2,3,4,5]
print_size(myarray)           # => The item's size is 5.
myfile = File::Stat.new("readme.txt")
print_size(myfile)            # => The item's size is 432.
```

It's called *duck* typing because an object that walks like a duck and quacks like a duck must be a duck. If so, then we should assume it is, in fact, a duck. Who are we to discriminate between a Mallard and a CanadianGeese? They both quack, right? Who are we to discriminate between a Man and a Woman? They both talk. We're expecting the object to do something, and we only have to ask: does it do that?

Duck typing is a trade off. You no longer have a compiler that will catch type errors upfront, but you do have fewer opportunities for errors. In a statically typed language, you'll repeat the type declaration in multiple places: class definition, variable declaration, constructor, method arguments, etc. If you need to refactor code, say splitting a class into an interface and separate implementation, maybe add a factory or write a decorator, you end up making type changes in multiple places, and you'll want a type checking compiler to help you minimize errors.

Why is that not a problem with Ruby? As you grow more familiar with Ruby you'll notice that you just don't have to declare types that often, or repeat them all over the place. You rarely need to separate interfaces and implementation classes, or to conjure factories and decorators. When you have fewer types to deal with, type checking is less of an issue. What you get in return is being able to do more with less code, which means fewer places for bugs to hide. Give it a shot, we'll doubt you'll miss type checking.

### **1.3.2 Keeping It Simple**

Ruby values simplicity. To be fair, all programming languages do, each striving for simplicity in its own way. They don't all achieve it in the same way. A language cannot enforce simplicity any more than it can keep your code bug free and as simple as possible to understand on its own terms, but it can certainly reward you for keeping things simple. It can also reward you for making things complex, often in the form of over-engineering.

As much as we hate to admit it, once you write a piece of software and release it to the world, the software becomes harder to change. Yet, it often needs to change: new features, switching databases, supporting more protocols, integrating with other systems. And so we plan for change. Each language has its own patterns that deal with change. In a statically typed language like Java, you will need to think these requirements upfront: once the implementation has been fixed, it is very hard to change. You tend to figure out the interfaces upfront, use factories liberally, allow for decorators, and so on. Because those changes are hard to do later on, you're better off doing them up front, just in case. Even those cases that will never happen. In their own way, statically typed languages reward you for over-engineering.

In contrast, being a dynamic language, you'll find that it's much easier to make local changes with Ruby, without affecting the rest of your code. When change is very easy to make, you don't have to plan as hard for every eventuality. In dynamic languages there's less need to design interfaces that are separate from the implementation, because it's possible to refactor the implementation without breaking code all around. You won't need to bury the business logic in layers of factories, decorators, listeners and anonymous classes. That might seem hard to imagine if you have a strong background with statically typed languages, but as you get comfortable with Ruby you'll notice it too. Ruby will reward you for keeping things simple, and saying no to code you don't need will reward you with quicker development and easier maintenance.

### **1.3.3 DRY efficiency**

Ruby is a DRY language. And we don't mean that Ruby is a bad date that can only tell lame jokes and talk about itself. Rather, Ruby helps you keep your code short and concise. DRY stands for: Don't Repeat Yourself. Syntactically, it's a very efficient language: you can express the same thing with less lines of code. As we know, computers are fast enough that more lines of code do not slow them down. But what about you? When it comes to debugging and maintaining, the more code you have to deal with, the harder it is to see what it does and find the problems that need fixing.

Listing 1.5 shows this with a simple example. The first style is Ruby, but you'll notice it looks similar to many other programming languages. The second style is the preferred way of doing it in Ruby: shorter and less repetitive (and even to that end, if this is the last value in a method, the return is superfluous). There's not a lot to this example, but imagine that you could do this throughout your code, eliminating thousands of lines of unnecessary cruft.

#### Listing 1.5 Two approaches to syntax

```
# The long way
record = Hash.new
record[:name] = "Dave"
record[:email] = "admin@net.com"
record[:phone] = "555-1235"
return record

# The Ruby way
return { :name=>"Dave", :email=>"admin@net.com", :phone=>"555-1235" }
```

Ruby's syntax has a lot of little features like this that end up giving you huge gains: blocks, iterators, open classes, and more. Part of these features, though, are due to Ruby's ability to do functional programming.

### 1.3.4 Functional programming

Ruby is an object-oriented language: it's objects all the way down. Like many other dynamic languages, functions are also first class citizens; couple that with outstanding support for closures, and it's easy to adopt a functional style of programming. But unlike more traditional functional programming languages, like LISP or Haskell, Ruby is easier to pick up, and can let you enjoy both worlds of functional and procedural style.

Why is functional programming so important? For one, it helps you write shorter and more concise code, some things are easier to express in functional style. Listing 2.2 shows an example of that. For another, the functional style leads to code that doesn't depend on state and has less side effects. Code like that is much easier to mold and refactor, and gives you more opportunities for reuse. It also makes it easier to build applications that scale.

The easiest way to build software that scales is using the "shared nothing" architecture. The less shared state you have to deal with, the easier it is to scale. Although Ruby comes with modern libraries that support threads, locks, mutexes and other concurrency mechanisms, it helps that you don't have to use them often. Perhaps you're familiar with the Google MapReduce algorithm that achieves unparalleled scalability by running tasks independently of each other. Yes, it's the same "map" as the one in Listing 1.6; the "reduce" part in Ruby is most often done using the *inject* method.

#### Listing 1.6 Three styles of iteration

```
# Code that iterates in order to map one array to another
application_names = []
for application in applications
  application_names << application.visible_name
names

# Code that maps one array to another
applications.map { |application| application.visible_name }
```

```
# An even shorter way to express it in Rails (and possibly Ruby 1.9)
applications.map &:visible_name
```

Code that has less dependencies is easier to run in parallel, taking advantage of modern multi-code CPUs, or when deploying on a cluster of servers. There are other ways to do it, but functional programming makes it extremely easy.

Now let's take a look at one of the most attractive features of Ruby; in covering these various facets of Ruby, we've been working our way towards one of the biggest features of the language: meta-programming.

## **1.4 Meta-programming**

We talked in the beginning about reframing solutions in terms of Ruby. Software development is the distance between your ideas, and applications you can actually use. The shorter that distance is, the more often you can travel down that path, the more you can carry with you. We're going to stop the analogies here, but the point we're trying to make is, you want to be able to translate your ideas into code quickly and easily.

Thinking out solutions in terms of Ruby helps you do that. But you can do even better if you bring Ruby closer to the way you imagine solutions. You can turn Ruby into a language that has a richer syntax and that is more expressive, so you can easily articulate what you want to do. We're talking, of course, about meta-programming.

Most programming languages are content being what they are. Some languages are more generic, but they force you to deal with more details. Other languages help you deal with a specific domain problem, but tend to be simple and inflexible. Languages like SQL, RuleML, XML Schema, Ant, CSS are these sorts of languages, and they are what we call domain-specific languages (which we'll talk about in the next section). But through meta-programming, Ruby allows you to have a mixture of both. You can extend Ruby with mini-languages that help you stay close to the problem you're solving, yet be able to use all the expressive power of Ruby.

This book is not an introduction to Ruby; what we want to talk to are the different technologies you can use to build business applications, integrate systems, and manage data. But there is one very powerful tool that will let you crank your developer volume all the way to 11, and give you the ability to solve problems with ease. And we think it's worthwhile to spend the next section introducing you to meta-programming and domain specific languages.

### **1.4.1 Getting Started With Meta-Programming**

"Programmers are devices for turning coffee into software" (at least the ones we know!). But jokes aside, software development is not about producing code, even if the recent trends toward outsourcing would lead you to believe that. As we've said before, software development is the distance between your problem and a usable piece of software to solve it. In fact, what will make you an even better software developer is knowing how to traverse this distance by writing less code. Not that code itself is evil, but the less code you have to deal with, the less time you spend debugging it, the less pain it is to maintain, the more time you have to solve new and interesting problems. And enjoy sunny days outside the office.

The software industry always looks for the Holy Grail, that one silver bullet that can turn ideas into code, without the "overhead" of software developers. We're not too worried about losing our

jobs. For all the hype and interest, software that writes software is the impossible dream, but software that writes software is also a wonderful tool, even if it only takes on part of the workload.

The thing is that software like that exists. Compilers that turn source code into machine code do that, they let us work with higher level languages. Source code generators and IDE wizards give us a head start by writing boilerplate code and skeletons. Then, even further up the ladder, there's *meta-programming*, writing code that writes code.

## Methods that Define Methods

If you come from a background in Java or C#, you learned that objects and classes are different in one fundamental way. Objects are mutable, so you can change them by calling methods on the objects, but in Java and C#, classes are immutable: their definition is written down in text and once compiled, cannot be changed. In Ruby, classes like objects are mutable. You can call a method on a class that will change the class definition.

Let's look at a simple example. We're dealing with a `Project` that has an `owner`, an attribute you can both get and set. Listing 1.7 shows two ways to express that. You can write a couple of methods, or you can call `attr_accessor`. By calling this method on the class, we allow it to change the class definition, and essentially define a method for getting the value of the instance variable `@owner`, and a method for setting it.

### Listing 1.7 Let `attr_accessor` define accessor methods on your class

```
# Define the accessor methods yourself
class Project
  def owner()                               #1
    @owner                                  |
  end                                       |

  def owner=(who)                            |
    @owner = who                            |
  end                                       |
end

# Let attr_accessor define them
class Project
  attr_accessor :owner                       #2
end
```

The above class definitions both do the same thing: define an attribute `owner` that you can get and set. But the second just uses a lot less code for us; this seems a little contrived now, but imagine having 12 or 15 attributes in a class. Then you're going from 40 or 50 lines of code down to 1. That's *huge* spread across an entire application. But how does it work? Let's take a look at an implementation of `attr_accessor` in Ruby (this is not *the* implementation from Ruby, but it has the same effect) in listing 1.8.

### Listing 1.8 A re-implementation of `attr_accessor`

```
class Module
  def attr_accessor (*symbols)              #1
    symbols.each do |symbol|               |
      class_eval %{                         #2
        def #{symbol}                      |

```

```

        @#{symbol}
      end

      def #{symbol}=(val)
        @#{symbol} = val
      end }
    end
  end
end

```

Using Ruby’s open classes, we re-open the class definition of Module (#1) and add a method, `attr_accessor`, to it. This method takes each provided symbol and executes a code block in the context of the class definition using `class_eval` #2. The `*_eval` family of methods is a family of methods you will become very familiar with in your meta-programming; check out Table 1.1 for a table of their usage.

**Table 1.1: The eval family of functions allows you to dynamically generate and execute code.**

Method	Usage
<code>eval(str)</code>	Evaluates a string of code.  <code>eval("puts 'I love eval!'")</code>
<code>instance_eval(str)</code> <code>instance_eval { }</code>	Evaluates a given string of code or code block, changing the receiver of the call to <code>self</code> .  <pre> class Hider   def initialize     @hidden = "Hello, world!"   end end  my_hidden = Hider.new my_hidden.instance_eval { puts @hidden } # =&gt; Hello, world! </pre>
<code>class_eval(str)</code> <code>class_eval { }</code>	Evaluates a given string of code or code block in the context of a class’s definition.  <pre> def printable_attribute(attr)   class_eval %{     def p_#{attr}       puts attr.to_s     end   } end  class Printer   attr_accessor :name   printable_attribute name end  my_printer = Printer.new my_printer.name = "LaserPrint 5000" my_printer.p_name # =&gt; LaserPrint 5000 </pre>

Now that we have our method added to the class definition, every instance of an object has this available to it.

Now let’s look at a more complex example. We’re further along the development of our application, and we realize there will be a lot of projects to manage. We want to help users by giving them a way to tag projects, so they can then find projects through tags and through related projects. So

we need a separate table to hold the tags, and we need to add methods to retrieve projects by tags, delete tag associations when a project is deleted, and so forth.

That's quite a lot of work: a lot of database work with inserting new tags, editing and deleting tags, not to mention search. We decide to re-invent the wheel some other time, and instead we download a plugin called `ActsAsTaggable`. Listing 1.9 shows how we use it.

**Listing 1.9 Using ActsAsTaggable to get a lot of features in one line of code**

```
class Project < ActiveRecord::Base
  acts_as_taggable
end
```

Why is this a more complex example when it's not longer than the previous one? Because `acts_as_taggable` adds a lot of functionality to our `Project` model that would take us several days to do right. Now our `Project` model has access to a set of tags in the database, along with methods to search with (e.g., `find_tagged_with`) baked right in. We can call `Project.find_tagged_with(:all => 'ruby')` and get an array of models that have the tag 'ruby.' It required very little code on our part: that is the power of meta-programming.

Ruby is flexible enough that in addition to defining new methods, you can extend existing methods, think of it as Aspect Oriented Programming baked into the language. It can also let you define complete classes. From the simple `Struct` which declares a bean-like class in one method call, to more complicated mechanisms, for example, creating a collection of model classes from your database schema.

## Implementing Methods Dynamically

This style of meta-programming happens at class definition, but meta-programming can also be used to dynamically alter objects. With languages that have static- (or early-) binding, you can only call a method if that method is part of the class definition. With languages that have dynamic- (or late-) binding, you can call any method on an object. The object then checks its class definition to decide how to respond to the method call.

This means that in Ruby, if the method is not part of the class definition, the object falls back by calling `method_missing`. Typically, `method_missing` will simply complain by throwing an exception, but you can override it to do more interesting things. Let's try to create an XML document (let's say an RSS feed for our projects) using `XML::Builder`; take a look at Listing 1.10.

**Listing 1.10 Building an RSS feed for our projects**

```
xml = XML::Builder.new
xml.rss "version"=>"2.0" do
  xml.channel do
    xml.title "Projects"
    xml.pubDate CGI.rfc1123_date(Time.now)
    xml.description "List of all our projects"
    for project in projects do
      xml.item do
        xml.title project.name
        xml.guid project_url(project), :permalink=>true
        xml.pubDate CGI.rfc1123_date(project.created_on)
        xml.description project.details
      end
    end
  end
end
```

```
end
end
```

If you're looking for the step where we use a DTD or XML Schema to generate source code, you won't find it. Builder uses `method_missing` to create elements based on the method name, attributes from the method arguments, and text nodes from the first argument, if it's a string. So, a very simplified version of that method might look like listing 1.11.

**Listing 1.11 A simplified look at XML::Builder's use of method\_missing**

```
def method_missing(sym, *args, &block)
  root_element = sym #1

  args.each do |arg| #2
    case arg
      when Hash #3
        build_elements(root_element)
      when String
        add_element(root_element)
    end
  end
end
```

Using Ruby's open classes, Builder overrides the method `method_missing`. Typically this method will just raise an exception since it is called when a method called on an object doesn't exist, but when overridden, can add interesting functionality. In this case, Builder takes the name of the missing method that is called #1 and the value provided #2 and makes elements out of them depending on the value's type #3. You'll likely find coding like this sprinkled throughout your favorite and most used libraries, including ActiveRecord.

Builder is one of those staple libraries you'll no doubt use quite often in your projects and an interesting library to analyze if you're interested in meta-programming. Doing so can help you come to a place where meta-programming becomes a tool that can help you build mini-languages that you can use to reframe business logic in terms of the problem domains rather than a language's syntax. We're talking, of course, about *domain specific languages*.

### 1.4.2 Domain Specific Languages

Functions, objects, libraries and frameworks all help you work at a higher level of abstraction, closer to the problem. If you look at the software landscape, you'll see a lot of different specialty domains. You can imagine languages, each of which is designed to solve a specific set of problems, by expressing solutions in a way that's easy and natural for that specific domain. Perhaps you never heard of domain specific languages (or DSLs for short) before reading this book, but you certainly have used them. Do any of the following names sound familiar? SQL, regular expressions, HTML, Make, WSDL, .htaccess, UML, CSS, Ant, XSLT, Bash. These are all domain specific languages.

Within the domain of relational databases, there needs to be a way to define new tables. There needs to be a way to query these tables and return the results, to create new records, to update and delete existing records. You could do those things by delving into low-level database APIs, creating the table structures directly, iterating over B-Tree indexes to fetch records, performing joins in memory and sorting the data yourself. In fact, many system developers used to do just that; nowadays, we use SQL.

But domain specific languages have a limit: it's very hard to create a new programming language that has good support for variables, expressions, functions, objects, and all the tooling around it. So domain specific languages tend to be very simple, static and inflexible. HTML, for example, can express rich multimedia content, including text, images, audio, and video. But if you want to do anything dynamic, like pull down menus, partial updates, drag & drop, you need a more generic programming language. You'll want to use JavaScript. On the other hand, because JavaScript is a generic programming language, it will take a mountain of statements to create a page using the DOM API. But what if you could mix a generic programming language with a domain specific language?

Let's look at XML Schema definitions for example. The XML Schema language was originally designed to validate XML documents. With it, you can express what a valid XML document looks like, so you can check XML documents against these rules before deciding whether or not to process them. Granted, it's much easier to use than iterating over the DOM and checking whether the current element is allowed to follow the previous element, and whether it has all the right attributes.

But, like most domain specific languages, XML Schema has its limits. For example, you can validate that a customer element has optional contact data, such as e-mail, phone or IM handle. But there's no easy way to require at least one of these elements to exist. There's no easy way to validate that all US addresses have a state, or that the zip code matches the actual address. But creating a DSL using a powerful host language (like Ruby), we could create something that looks like listing 1.12.

**Listing 1.12: A validation domain specific language example**

```
contact_information_verification do |the_persons|
  the_persons.name.is_required

  the_persons.address.is_required
  the_persons.address.must_be(10).characters_long

  the_persons.phone.must_be(10).characters_long

  the_persons.im_handle.must_not_be(in_existing_accounts)
end
```

Using this, we have a nice set of readable rules that can be updated by anyone (probably even your secretary!), which will generate or execute the validations we need. DSLs can bring the same simplicity and abstraction to a lot of your specific problems. Think about code that expresses business rules or composes tasks, and you'll find many opportunities to simplify and reduce noise. DSLs don't just help your secretary or business manager keep your code up, but it can also help you as a developer keep yourself sane. Using Ruby's metaprogramming capabilities, you can build these sorts of solid, literate tools and fluent interfaces.

**EMBEDDED DSLS**

What we've created here is not *technically* a DSL in the purest, scientific sense, but an embedded DSL (EDSL) – a domain specific language that's embedded inside a host language, in this case Ruby. Throughout this book, we'll show different Ruby libraries and tools that include their own mini-languages, all of which are EDSLs.

### **1.4.3 Refining your meta-programming**

If you never worked with meta-programming before, you're probably asking some of the same questions we asked ourselves: Where do I start? What's a good meta-programming style? Should I

worry about methods that are not explicitly defined? How do I know my code works, can I even test it?

The place to start is with the problem you solve over and over again. When you see a lot of repetition in your code, when you find yourself getting bogged down by details, that's the right time to simplify things. Meta-programming helps you simplify by creating an easier, more expressive way to write code. There's no point in writing a DSL that you will only use once, but it pays several times over when you use it repeatedly.

You could let your imagination run wild and use meta-programming to solve problems you think you'll have someday, but you'll quickly realize those are just mental exercises. Instead, look for patterns you have in your code right now, for practices and idioms you use often, and use meta-programming to simplify those. It's called extraction. Extraction comes from what you need and use, not what you could imagine doing some day. You'll notice that Rails, Rake, RSpec and many of the other frameworks we cover in this book all came from extractions, from looking at existing solutions and finding better, easier ways to write those.

The best way to write a DSL, is to practice "intentional programming". Step away from the code for a minute and ask yourself, "If I were not limited by the language I use right now, if I could express the solution itself in some other language, what would my code look like?" Try and write that code as if that language already exists. You've just defined a DSL. Now you have to go and implement it, but the hardest part is behind you.

So how do you know you're successful? A good DSL has two interesting qualities. The first, you'll want to use it. You know you're successful when, out of all the possible things you can do that day, just after coffee, you decide to write code using that DSL.

The second quality of a good DSL is that you can throw away the documentation and still use it. It's called "the element of least surprise". When the code comes naturally to you, it's easier to write and less painful to maintain. When we work with Rails, we never have to stop and ask "How do we access the order\_on field?" We know that in Rails, it's accessed with the order\_on method. We never have to ask "How do I find an employee record by their name?" We know to use find\_by\_name. The rule of least surprise is one of those "I'll know it when I see it": just ask your team mates what they expect to happen. And if you follow our advice for intentional programming, you'll find that it's an easy principle to follow.

Using method\_missing and a few other fun meta-programming tricks, you may work yourself into a position where you have a lot of methods that aren't explicitly defined. How do you deal with that? You document, of course. And if you follow the rule of least surprise, you have little to worry about: other developers will intuitively know which methods to expect, when it all makes sense.

So, what about testing? Don't the dynamic features make it harder to test? Testing meta-programming is no different from testing any other piece of code. There's nothing magical about it, no hidden traps. You can use any faculty available for testing, in fact, in the next section we're going to talk about testing at length, and cover a testing framework called Rspec, which uses meta-programming to help you write tests efficiently by expressing the expected behavior of your software.

## **1.5 Summary**

Ruby has gained a lot of popularity in recent times for being a simpler, expressive alternative to other contemporary languages. This success is partially due to the emergence of Rails as a tangible tool that developers can use to develop production quality software that's still developer friendly. Rails owes

this friendliness to Ruby, which offers a lot of features, such as meta-programming, that make Ruby more expressive and even fun.

As more and more developers pick up Ruby, they will want to use their current programming paradigms like aspect-oriented programming or declarative programming and development practices such as Scrum or test-driven development. Ruby has a number of libraries to help the process for many of these (such as Ruleby or AspectR). In the next chapter, we'll concentrate on the most prevalent of these, test-driven development, and look at testing your Ruby code using the built-in testing library and some related third-party tools and libraries.