

Chapter 1

Welcome to the World of Ruby on Rails

In This Chapter

- ▶ Understanding the need for agile software development
 - ▶ Discovering Ruby's role in agile development
 - ▶ Finding out how Rails fits in
-

Once upon a time, there were three little programmers. The programmers wrote code for the World Wide Web — code to give users access to a company's database.

The first programmer was in a hurry to write her code. She wrote simple code as quickly as she could. The second programmer wasn't quite in such a hurry. She used the traditional Waterfall methodology — a multistep process involving analysis, design, coding, testing, and deployment. The third programmer was careful and industrious. She used a heavyweight persistence framework such as Enterprise JavaBeans. She built her software to cover every possible contingency and to accommodate any future need.

As you might expect, this story has a big bad wolf. The wolf might have been a manager, a client paying for the software's creation, or a customer attempting to access the company's Web site. The wolf went in reverse order, visiting the careful and industrious programmer's Web site first.

Unfortunately, the wolf couldn't log onto the industrious programmer's site. Instead, he got the message: "This site is under construction." The careful, industrious programmer had completed only half of her work. The heavyweight persistence framework was difficult to learn and burdensome to use. Needless, to say, the wolf huffed and he puffed, and he blew the Web site down.

So the wolf visited the second programmer's Web site. The site was up and running, but certain aspects of the site didn't meet the wolf's needs. In following the Waterfall methodology, the second programmer had carefully planned every aspect of the project before beginning to write the code. But by the time the code was ready for testing, the project's requirements had shifted.

The second programmer was aware of her Web site's deficiencies. Through extended testing and use, she had learned that the original requirements were obsolete. But with all the code in place, the second programmer couldn't easily make major changes. All she could do was fix bugs and make the code run a bit faster. She promised that she'd update the requirements for version 2.0 of the system. But the wolf was impatient. He huffed and he puffed, and he blew the Web site down.

In desperation, the wolf visited the first programmer's Web site. She had built the site quickly and easily, using Ruby on Rails. In fact, her first prototype had been up and running in two days. Her co-workers had tested the prototype, critiqued the prototype's features, and told her what they expected in the next prototype.

The next prototype was ready sooner than anyone expected. Once again, co-workers tested the prototype, suggested improvements, and helped the programmer to refine her evolving requirements.

After several brief rounds of coding and testing, the Web site was ready for public use. The wolf enjoyed visiting the site because the site's look and feel reflected the way it had been designed. The site was nimble, intelligent, and easy to use. The site did the kinds of things the wolf wanted it to do because the programmer had gotten feedback on each prototype. Everyone was happy . . . for a while anyway.

To repay the Ruby on Rails programmer, the wolf offered to repair her house's leaking roof. Unfortunately, the wolf had a nasty accident. While he was working on the roof, he fell into the chimney and landed directly into a pot of boiling water. Goodbye, wolf!

But the Ruby on Rails programmer was happy. She had created a great Web site. And with all the time she'd saved using Ruby on Rails, she was able to climb up to the roof and repair the leak herself.

The end.

The Software Development Process

The world changes quickly. Ten years ago, when I taught programming to computer professionals, I wore a suit and a tie. Last month I taught the same course wearing a polo shirt and khakis.

This tendency for things to change goes way back. In the 1960s, programmers and managers noticed that commercial software tended to be very buggy. They analyzed large projects created by big businesses. They saw software development efforts going past deadline and over budget. They saw finished products that were terribly unreliable. Most computer code was difficult to test and impossible to maintain.

So they panicked.

They wrote books, magazine articles, and scholarly papers. They theorized. They devised principles, and they arrived at various conclusions.

After years of theorizing, they founded the discipline known as *software engineering*. The goal of software engineering is to discover practices that help people write good code. As disciplines go, software engineering is pretty good stuff. Software engineering encourages people to think about the way they create software. And when people think about the way they work, they tend to work better.

But in the 1970s, software engineering focused on methodologies. A *methodology* is a prescribed set of practices. Do this, then do that, and finally, do the other thing. When you're finished, you have a big software system. But do you have a useful software system?

In 1979, I worked briefly for a company in Milwaukee. On the day I arrived, the team manager pointed to the team's methodology books. The books consisted of two monstrous volumes. Together the volumes consumed about six inches of bookshelf. I remember the team manager's words as he pointed a second time to the books. "That's what we use around here. Those are the practices that we follow."

I spent several months working for that company. In all those months, no one ever mentioned the methodology books again. I would have cracked the books open out of curiosity. But unfortunately, excessive dust makes me sneeze. Had I found anyone on the team who knew the methodology, I probably

would have learned how ponderous the methodology can be. No one wanted to wade through hundreds of pages of principles, rules, and flow diagrams. And if anyone did, they'd read about rigid systems — systems that encourage programmers to follow fixed procedures — systems that don't encourage programmers to listen, to adjust, or to change.

Agility

In 2001, a group of practitioners created the *Manifesto for Agile Software Development* (www.agilemanifesto.org). The Manifesto's signatories turned their backs on the methodologies of the past. Instead, they favored a nimble approach. Their principles put "individuals and interactions over processes and tools," and put "responding to change over following a plan." Best of all, they declared that "Simplicity — the art of maximizing the amount of work not done — is essential." According to these practitioners, the proof of the pudding is in the result. A process that doesn't end in a worthwhile result is a bad process, even if it's an orderly, well-established process.

The Agile Manifesto's signatories aren't opposed to the discipline of software engineering. On the contrary, they believe firmly in the science of software development. But they don't believe in unnecessary paperwork, required checklists, and mandatory diagrams. In other words, they don't like horse pockey.

Databases and the World Wide Web

By 2001, many businesses faced an enormous problem. Computers were no longer islands unto themselves. Customers visited Web sites, ordered goods, read headlines, updated records, posted comments, and downloaded songs. At one end was a Web browser; at the other end was a database. In between was lots of network plumbing. The problem was to move data from the browser to the database, and from the database to the browser. The movement must be efficient, reliable, and secure.

Imagine millions of people working on the same problem — moving data between a Web browser and a database. If everyone works independently, then millions of people duplicate each others' efforts. Instead of working independently, why not have people build on other people's work? Create a software framework for connecting Web browsers to databases. Provide hooks into the software so that people can customize the framework's behavior. An online order system uses the framework one way, and a social networking site uses the framework in its own, completely different way.

Throwing frameworks at the problem

By 2004, there wasn't just one framework for solving the Web/database problem. There were dozens of frameworks. New frameworks, with names such as Enterprise JavaBeans, Spring, Hibernate, and .NET, tackled pieces of the problem.

But most of the aforementioned frameworks had a serious deficiency. They didn't lend themselves to agile software development. Software created with one of these frameworks was fairly rigid. Planning was essential. Changes were costly.

What the world needed was a different framework — a framework for agile developers. The world needed a language that didn't put programmers in a box. The world needed software that could shift with a user's shifting needs. Let the major corporations use the older, heavyweight frameworks. An entrepreneurial company thrives with a more versatile framework. A small-to-medium-size company needs Ruby on Rails.

Along Comes Ruby on Rails

Think about your native language — the language you speak at home. Divide the language into two styles. You use one style when you speak to a close friend. ("Hi, buddy.") You use another, more formal style when you write to a potential employer ("Dear Sir or Madam . . .").

Talking to a close friend is an agile activity. You listen intently, but occasionally you interrupt. If your friend says something intriguing, you take time out to ask for more details. You don't try to impress your friend. You tune carefully to your friend's mood, and the friend tunes to your mood.

In contrast, writing a business cover letter is not an agile activity. You don't get feedback as you write the letter. You try to guess what the potential employer wants you to say, but you can never be sure. You use a formal writing style in case the employer is a stodgy old coot.

Now imagine using a formal style to speak to your friend. "If you have any questions about our next meeting at Kelly's Tavern, please don't hesitate to call me at the phone number on this napkin. I look forward to hearing from you soon. Yours truly, *et cetera, et cetera*." Using formal language with your friend would slow the conversation to a crawl. You wouldn't pop your eyes open when you heard some juicy gossip. Instead, you'd plan each sentence carefully. You'd think about subject/verb agreement, hoping you didn't offend your friend with an awkward phrase or with some inappropriate slang.

Language isn't a neutral medium of expression. Language influences the nature of the message. A free-flowing style encourages free-flowing thought. In the same way, a flexible programming language complements an agile software development process.

Why Ruby?

Ruby is a computer programming language. You might be familiar with Basic, Java, C++, or some other programming language. In certain ways, all these languages are the same. They all provide ways for you to give instructions to a computer. "Move this value from that memory location to that other location on your hard drive." A computer language is a way of expressing instructions in a precise, unambiguous manner.

What makes Ruby different from so many other computer programming languages? In what way does Ruby support agile development?

Here's the answer: Ruby is a dynamically typed, interpreted, reflective, object-oriented language. That's a great answer, but what does it mean?

Ruby is dynamically typed

In many languages, you have to declare each variable's type. You write

```
int date;  
date = 25092006;
```

The first line tells the computer that the `date` must store an integer — a whole number — a number without a decimal point — a number like 25092006. Later in the same program, you might write

```
date = "September 25, 2006";
```

But the computer refuses to accept this new line of code. The computer flags this line with an error message. The value "September 25, 2006" isn't an integer. (In fact, "September 25, 2006" isn't a number.) And because of the `int date;` line, the non-Ruby program expects `date` to store an integer.

The word `int` stands for a type of value. In a *statically* typed language, a variable's type doesn't change.

In contrast, Ruby is *dynamically* typed. The following lines form a complete, valid Ruby program:

```
date = 25092006  
date = "September 25, 2006"
```

(Yes, this program doesn't do anything useful, but it's a program nevertheless.)

Ruby's variables can change from being integers to being decimal values, and then to being strings or arrays. They change easily, without any complicated programming techniques. This flexibility makes Ruby a good language for agile software development.

Ruby is interpreted

Many commonly used programming languages are *compiled*. When the computer compiles a program, the computer translates the program into a very detailed set of instructions (a set more detailed than the set that the programmer originally writes).

So picture yourself developing code in a compiled language. First you write the code. Then you compile the code. Then you run the code. The code doesn't run exactly the way you want it to run, so you modify the code. Then you compile again. And then you run the code again. This cycle takes place hundreds of times a day. "Modify, compile, run." You get tired of saying it inside your head.

In contrast to the compiled languages, Ruby is *interpreted*. An interpreted language bypasses the compilation step. You write the code, and then you run the code. Of course you don't like the results. (That's a given.) So you modify and rerun the code. The whole cycle is much shorter. A piece of software (the Ruby *interpreter*) examines your code and executes that code without delay.

Which is better — compiled code or interpreted code? Believe it or not, the answer depends on your point of view. A computer executes compiled code faster than interpreted code. But as computer processing power becomes cheaper, the speed of execution is no longer such an important issue.

So step back from the processing speed issue and think about the speed of software development. With a compiled language, each modify-compile-run cycle is three steps long, compared with the two-step modify-run cycle in an interpreted language such as Ruby. But what's the big deal? How long can an extra compilation step possibly take?

The answer is that compilation can slow you down. Compilation can be time consuming, especially on a very large, complex project. Even a two-second compilation can be annoying if you perform the cycle several hundred times a day.

But aside from the time factor, the compilation step distances the programmer from the run of a program. Imagine writing a program in a compiled language, say in C++. The computer compiles your program to get a more detailed set of instructions. This detailed set of instructions isn't the same as your original program. It's a translation of your instructions. Instead of executing your program, the computer executes a translation.

Little to nothing gets lost in translation. But the fact that the computer doesn't run your original code makes a difference in the way you think about the development cycle. The immediate, hands-on feeling of an interpreted language gives an extra lift to the agile development mindset.

Ruby is reflective

A Ruby program can reflect upon its own code, like a philosopher reflecting on his or her own life. More specifically, a Ruby program can turn a string of characters into executable code and can do this somersault during the run of a program. Listing 1-1 contains an example:

Listing 1-1: Defining a Database Table

```
print "Enter some text: "
STDOUT.flush
text_input = gets
puts

print "You entered: "
print text_input
puts

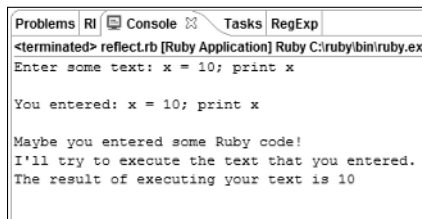
print "Maybe you entered some Ruby code!\n"
print "I'll try to execute the text that you entered.\n"
print "The result of executing your text is "
eval text_input
```

Figures 1-1 and 1-2 show two different runs of the code in Listing 1-1. In each run the code prompts you to type some text. Ruby does two things with whatever text you type:

- ✓ Ruby echoes the text (displays the text a second time on the screen).
- ✓ Ruby interprets your text as Ruby code and executes the code if possible.

The second step (reinterpreting text as code) is difficult to do in other programming languages. Ruby makes it easy to reinterpret text as code, and this ease makes life better for computer programmers.

Figure 1-1:
A run of the
code in
Listing 1-1.



```
<terminated> reflect.rb [Ruby Application] Ruby C:\ruby\bin\ruby.exe
Enter some text: x = 10; print x

You entered: x = 10; print x

Maybe you entered some Ruby code!
I'll try to execute the text that you entered.
The result of executing your text is 10
```

Figure 1-2:
Running the
code with
more com-
plicated
input.

```

Problems | Console | Tasks | RegExp
<terminated> reflect.rb [Ruby Application] Ruby C:\ruby\bin\ruby.exe: reflect.rb
Enter some text: prod = 1; 5.times { |x| print prod *= x + 1, ' ' }

You entered: prod = 1; 5.times { |x| print prod *= x + 1, ' ' }

Maybe you entered some Ruby code!
I'll try to execute the text that you entered.
The result of executing your text is 1 2 6 24 120

```

Ruby is object-oriented

I describe object-oriented programming (OOP) in Chapter 6. So I don't want to spoil the fun in this chapter. But to give you a preview, object-oriented programming centers around nouns, not verbs. With object-oriented programming, you begin by defining nouns. Each *account* has a name and a balance. Each *customer* has a *name*, an *address*, and one or more *accounts*.

After describing the nouns, you start applying verbs. *Create* a new account for a particular customer. *Display* the account's balance. And so on.

Since the late 1980s, most commonly used programming languages have been object oriented. So I can't claim that Ruby is special this way. But Ruby's object-oriented style is more free-form than its equivalent in other languages. Again, for more details on object-oriented programming in Ruby, see Chapter 6.

Why Rails?

Rails is an add-on to the Ruby programming language. This add-on contains a library full of Ruby code, scripts for generating parts of applications, and a lot more.

The name *Ruby on Rails* is an inside joke. Since the year 2000, teams of Java programmers have been using a framework named *Struts*. The Struts framework addresses many of the problems described in this chapter — Web development, databases, and other such things. But the word *strut* means something in the construction industry. (A *strut* is a horizontal brace, and a sturdy one at that.) Well, a *rail* is also a kind of horizontal brace. And like *Ruby*, the word *Rail* begins with the letter *R*. Thus the name *Ruby on Rails*.



In spite of the name *Ruby on Rails*, you don't add Ruby on top of Rails. Rather, the Rails framework is an add-on to the Ruby programming language.

The following fact might not surprise you at all. What separates Rails from Struts and other frameworks is agility. Other frameworks used to solve the Web/database problem are heavy and rigid. Development in these other frameworks is slow and formal. In comparison, Rails is lightweight and nimble.

Author and practitioner Curt Hibbs claims that you can write a Rails application in one-tenth the time it takes to write the same application using a heavyweight framework. Many people challenge this claim, but the fact that Hibbs is willing to make the claim says something important about Rails.

Rails is built on two solid principles: convention over configuration, and Don't Repeat Yourself (DRY).

Convention over configuration

A Web application consists of many parts, and you can go crazy connecting all the parts. Take one small example. You have a variable named `picture` in a computer program, and you have a column named `image` in a database table. The computer program fetches data from the `image` table column and stores this data in the `picture` variable. Then the program performs some acrobatics with the `picture` variable's data. (For example, the program displays the picture's bits on a Web page.)

One way to deal with an application's parts is to pretend that names like `picture` and `image` bear little relation to one another. A programmer stitches together the application's parts using a *configuration file*. The configuration file encodes facts such as “variable `picture` reads data from column `image`,” “variable `quotation` reads data from column `stock_value`,” and “variable `comment_by_expert` reads data from column `quotation`.” How confusing!

With dozens of names to encode at many levels of an application, programmers spend hours writing configuration files and specifying complex chains of names. In the end, errors creep into the system, and programmers spend more hours chasing bugs.

Rails shuns configuration in favor of naming conventions. In Rails, a variable named `image` matches automatically with a column of the same name in the database table. A variable named `Photo` matches automatically with a table named `photos`. And a variable named `Person` matches automatically with a table named `people`. (Yes, Rails understands plurals!)

In Rails, most configuration files are completely unnecessary. You can create configuration information if you want to break Ruby's naming conventions. But if you're lucky, you seldom find it necessary to break Ruby's naming conventions.

Don't Repeat Yourself (DRY)

Another important Rails principle is to avoid duplicating information. A traditional program contains code describing database tables. The code tells the rest of the program about the structure of the tables. Only after this descriptive code is in place can the rest of the program read data from the database.

But in some sense, the description of a database is redundant. A program can examine a database and automatically deduce the structure of the database's tables. Any descriptive code simply repeats the obvious. "Hey, everyone. There's a gorilla in the room. And there's an `image` column in the `photos` database table." So what else is new?

In computer programming, repetition is bad. For one thing, repetition of information can lead to errors. If the description of a database is inaccurate, the program containing the description doesn't work. (My HMO asks for my address on every claim form. But my address hasn't changed in the past ten years. Occasionally, the folks who process the claim forms copy my address incorrectly. They mail a reimbursement check to the wrong house. Then I make ten phone calls to straighten things out. That's a danger of having more than one copy of a certain piece of information.)

Aside from the risk of error, the duplication of information means more work for everyone. With traditional database programming, you must track every decision carefully. If you add a column to a database table, you must update the description of the database in the code. The updating can be time-consuming, and it discourages agility. Also, if each change to a database table requires you to dive into your code, you're less likely to make changes. If you avoid changes, you might not be responding to your customer's ever-changing needs.

Let's Get Going

You can read this chapter's lofty paragraphs until you develop a throbbing headache. But the meaning behind these paragraphs might be somewhat elusive. Do you feel different when you switch from C++ or Java to programming in Ruby? Does Rails really speed up the development cycle? Can you create an application in the time it takes to find a Starbucks in Manhattan? If you find these questions intriguing, please read on.

