Introduction

The way that can be named is not the true Way.

—Lao Tse, *Tao Te Ching*

The title of this book is *The Ruby Way*. This is a title that begs for a disclaimer.

It has been my aim to align this book with the philosophy of Ruby as well as I could. That has also been the aim of the other contributors. Credit for success must be shared with these others, but the blame for any mistakes must rest solely with me.

Of course, I can't presume to tell you with exactness what the spirit of Ruby is all about. That is primarily for Matz to say—and I think even he would have difficulty communicating all of it in words.

In short, *The Ruby Way* is only a book, but the Ruby Way is the province of the language creator and the community as a whole. This is something difficult to capture in a book.

Still, I have tried in this introduction to pin down a little of the ineffable spirit of Ruby. The wise student of Ruby will not take it as authoritative.

How This Book Works

You won't learn Ruby from this book. There is relatively little in the way of introductory or tutorial information. If you are totally new to Ruby, I suggest you start with another book.

Having said that, programmers are a tenacious bunch, and I grant that it might be possible to learn Ruby from this book. Chapter 1, "Ruby in Review," does contain a brief introduction and some tutorial information.

Chapter 1 also contains a fairly comprehensive "gotcha" list (which has been hard to keep up-to-date). The usefulness of this list will vary widely from one reader to another because we cannot all agree on what is intuitive.

This book is largely intended to answer questions of the form of "How do I…?" As such, you can expect to do a lot of skipping around. I'd be honored if everyone read every page from front to back, but I don't expect that. It's more my expectation that you will browse the table of contents in search of techniques you need or topics you find interesting.

Some areas this book covers are very elementary. That's because people vary in background and experience; what is obvious to one person may not be to another. I have tried to err on the side of completeness. On the other hand, I have tried to keep the book at a reasonable size (obviously a competing goal).

This book can be viewed as a sort of "inverted reference." Rather than looking up the name of a method or a class, you will look things up by function or purpose. For example, the `String` class has four methods for manipulating case: `capitalize`, `upcase`, `downcase`, and `swapcase`. In a reference work, these would quite properly be listed alphabetically, but in this book they are all listed together.

Of course, in striving for completeness, I have sometimes wandered onto the turf of the reference books. In many cases, I have tried to compensate for this by offering more unusual or diverse examples than you might find in a reference.

I have tried for a very high code-to-commentary ratio. Overlooking the initial chapter, I think I've achieved this. Writers may grow chatty, but programmers always want to see the code. (If not, they *should* want to.)

The examples here are sometimes very contrived, for which I must apologize. To illustrate a technique or principle in isolation from a real-world problem can be very difficult. However, the more complex or "high level" the task was, the more I attempted a real-world solution. Therefore, if the topic is concatenating strings, you may find an unimaginative code fragment involving `"foo"` and `"bar"`, but when the topic is parsing XML, for example, you will usually find a much more meaningful and realistic piece of code.

This book has two or three small quirks to which I'll confess up front. One is the avoidance of the "ugly" Perl-like global variables such as `$_` and others. These are present in Ruby, and they work fine; they are used daily by most or all Ruby programmers. However, in nearly all cases, their use can be avoided, and I have taken the liberty of omitting them in most of the examples.

Another quirk is that I avoid using standalone expressions when they don't have side effects. Ruby is expression oriented, and that is a good thing. I have tried to take advantage of that feature. However, in a code fragment, I prefer not to write expressions that merely return a value that is not usable. For example, the expression `"abc" + "def"` can illustrate string concatenation, but I would write something like `str = "abc" + "def"` instead. This may seem wordy to some, but it may seem more natural to you if you are a C programmer who really notices when functions are void or non-void (or an old-time Pascal programmer who thinks in procedures and functions).

My third quirk is that I don't like the "pound" notation to denote instance methods. Many Rubyists will think I am being verbose in saying "instance method `crypt` of class `String`" rather than saying "`String#crypt`," but I think no one will be confused.

I've added a feature to this book that I believe will be very convenient, and I hope you agree. There is a "keywords" line for every section in the table of contents, helping to answer questions such as "Does this section mention XYZ?" without you having to turn to that page. This, in my opinion, brings some of the usefulness of the index into the table of contents.

I have tried to include "pointers" to outside resources whenever appropriate. Time and space did not allow putting everything into this book that I wanted, but I hope I have partially made up for that by telling you where to find related materials. The Ruby Application Archive on the Web is probably the foremost of these sources; you will see it referenced many times in this book.

Here, at the front of the book, is usually where you'll find a gratuitous reference to the typefaces used for code and how to tell code fragments from ordinary text. However, I won't insult your intelligence; you've read computer books before.

And now a word about personal pronouns. Here in the introduction, I've used the singular first person consistently, but throughout most of the book, I've used *we*, in the manner of kings and editors. This is not (just) my pompous nature revealing itself; it's a very real concession to the fact that a little over 10 percent of this book was written by other people. (Most readers skip the acknowledgements in a book. Go read them now. They're good for you, like vegetables.)

What Is the Ruby Way?

Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all.

—Douglas Adams, *Dirk Gently's Holistic Detective Agency*

What do we mean by the Ruby Way? My belief is that there are two related aspects: One is the philosophy of the *design* of Ruby; the other is the philosophy of its usage. It is natural that design and use should be interrelated, whether in software or hardware; why else should there be such a field as ergonomics? If I build a device and put a handle on it, it is because I expect someone to grab that handle.

Ruby has a nameless quality that makes it what it is. We see that quality present in the design of the syntax and semantics of the language, but it is also present in the programs written for that interpreter. Yet, as soon as we make this distinction, we blur it.

Clearly, Ruby is not just a tool for creating software but is a piece of software in its own right. Why should the workings of Ruby *programs* follow laws different from those that guide the workings of the *interpreter*? After all, Ruby is highly dynamic and extensible. There might be reasons why the two levels should differ here and there—probably for accommodating to the inconvenience of the real world. However, in general, the thought processes can and should be the same. Ruby could be implemented in Ruby, in true Hofstadter-like fashion, although it is not at the time of this writing.

We don't often think of the etymology of the word *way*, but there are two important senses in which it is used. On the one hand, it means a *method* or *technique*, but it can also mean a *road* or *path*. Obviously, these two meanings are interrelated, and I think when I say "the Ruby Way," I mean both of them.

So what we are talking about is a thought process, but it is also a path that we follow. Even the greatest software guru cannot claim to have reached perfection, but only to follow the path. And there may be more than one path, but here I can only talk about one.

The conventional wisdom says that "form follows function." And the conventional wisdom is, of course, conventionally correct. However, Frank Lloyd Wright (speaking of his own field) once said, "Form follows function—that has been misunderstood. Form and function should be one, joined in a spiritual union."

What did Wright mean? I would say that this truth is not something you learn from a book, but from experience.

However, I would argue that Wright expressed this truth elsewhere in pieces easier to digest. He was a great proponent of simplicity, saying once, "An architect's most useful tools are an eraser at the drafting board and a wrecking bar at the site."

So one of Ruby's virtues is simplicity. Shall I quote other thinkers on the subject? According to Antoine de St. Exupery, "Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away."

But Ruby is a complex language. How can I say that it is simple?

If we understood the universe better, we might find a "law of conservation of complexity"—a fact of reality that disturbs our lives like entropy so that we cannot avoid it but can only redistribute it.

And that is the key. We can't avoid complexity, but we can push it around. We can bury it out of sight. This is the old "black box" principle at work; a black box performs a very complex task, but it possesses simplicity *on the outside*.

If you haven't already lost patience with my quotations, a word from Albert Einstein is appropriate here: "Everything should be as simple as possible, but no simpler."

So, in Ruby we see simplicity embodied from the programmer's view (if not from the view of those maintaining the interpreter). Yet we also see the capacity for compromise. In the real world, we must bend a little. For example, every entity in a Ruby program should be a true object, but certain values such as integers are stored as immediate values. In a tradeoff familiar to computer science students for decades, we have traded elegance of design for practicality of implementation. In effect, we have traded one kind of simplicity for another.

What Larry Wall said about Perl holds true: "When you say something in a small language, it comes out big. When you say something in a big language, it comes out small." The same is true for English. The reason that biologist Ernst Haeckel could say "Ontogeny recapitulates phylogeny" in only three words was that he had these powerful words with highly specific

meanings at his disposal. We allow inner complexity of the language because it enables us to shift the complexity away from the individual utterance.

I would state this guideline this way: *Don't write 200 lines of code when 10 will do.*

I'm taking it for granted that brevity is generally a good thing. A short program fragment will take up less space in the programmer's brain; it will be easier to grasp as a single entity. As a happy side effect, fewer bugs will be injected while the code is written.

Of course, we must remember Einstein's warning about simplicity. If we put it too high on our list of priorities, we will end up with code that is hopelessly obfuscated. Information theory teaches us that compressed data is statistically similar to random noise; if you have looked at C or APL or regular expression notation—especially badly written—you have experienced this truth firsthand. Simple, but not too simple; that is the key. Embrace brevity, but do not sacrifice readability.

It is a truism that both brevity and readability are good. However, there is an underlying reason for this—one so fundamental that we sometimes forget it. The reason is that *computers exist for humans, not humans for computers*.

In the old days, it was almost the opposite. Computers cost millions of dollars and ate electricity at the rate of many kilowatts. People acted as though the computer were a minor deity and the programmers were humble supplicants. An hour of the computer's time was more expensive than an hour of a person's time.

When computers became smaller and cheaper, high-level languages also became more and more popular. These were inefficient from the computer's point of view but efficient from the human perspective. Ruby is simply a later development in this line of thought. Some, in fact, have called it a *VHLL* (very high-level language); although this term is not well-defined, I think its use is justified here.

The computer is supposed to be the servant, not the master; and, as Matz has said, a smart servant should do a complex task with a few short commands. This has been true through all the history of computer science. We started with machine languages and progressed to assembly language and then to high-level languages.

What we are talking about here is a shift from a machine-centered paradigm to a human-centered one. In my opinion, Ruby is an excellent example of human-centric programming.

I'll now shift gears a little. There was a wonderful little book from the 1980s called *The Tao of Programming*, by Geoffrey James. Nearly every line is quotable, but I'll repeat only one: "A program should follow the 'Law of Least Astonishment.' What is this law? It is simply that the program should always respond to the user in the way that astonishes him least." (Of course, in the case of a language interpreter, the *user* is the programmer.)

I don't know whether James coined this term, but his book was my first introduction to the phrase. This is a principle that is well known and often cited in the Ruby community, although it is usually called the *Principle of Least Surprise*, or POLS. (I myself stubbornly prefer the acronym LOLA.)

Whatever you call it, this rule is a valid one, and it has been a guideline throughout the ongoing development of the Ruby language. It is also a useful guideline for those who develop libraries or user interfaces.

The only problem, of course, is that different people are surprised by different things; there is no universal agreement on how an object or method "ought" to behave. However, we can strive for consistency and to justify our design decisions, and each person can train his own intuition.

No matter how logically constructed a system may be, your intuition needs to be trained. Each programming language is a world unto itself, with its own set of assumptions; and human languages are the same. When I took German, I learned that all nouns were capitalized; but the word *deutsch* was not. I complained to my professor; after all, this was the *name* of the language, wasn't it? And he smiled and said, "Don't fight it."

What he taught me was to *let German be German*—and by extension, that is good advice for anyone coming to Ruby from some other language. Let Ruby be Ruby. Don't expect it to be Perl, because it isn't; don't expect it to be LISP or Smalltalk, either. On the other hand, Ruby has common elements with all three of these. Start by following your expectations, but when they are violated, don't fight it. (Unless Matz agrees it's a needed change.)

Every programmer today knows the orthogonality principle (which would better be termed the *orthogonal completeness* principle). Suppose we have an imaginary pair of axes with a set of comparable language entities on one and a set of attributes or capabilities on the other. When we talk of "orthogonality," we usually mean that the space defined by these axes is as "full" as we can logically make it.

Part of the Ruby Way is to strive for this orthogonality. An array is in some ways similar to a hash, so the operations on each of them should be similar. The limit is reached when we enter the areas where they are different.

Matz has said that "naturalness" is to be valued over orthogonality. But to fully understand is natural, and what is not may take some thinking and some coding.

Ruby strives to be friendly to the programmer. For example, there are aliases or synonyms for many method names; `size` and `length` will both return the number of entries in an array. The variant spellings `indexes` and `indices` both refer to the same method. Some consider this sort of thing to be an annoyance or "anti-feature," but I consider it a good design.

Ruby strives for consistency and regularity. There is nothing mysterious about this; in every aspect of life, we yearn for things to be regular and parallel. What makes it a little more tricky is learning when to violate this principle.

For instance, Ruby has the habit of appending a question mark (`?`) to the name of a predicate-like method. This is well and good; it clarifies the code and makes the namespace a little more manageable. But what is more controversial is the similar use of the exclamation point in marking methods that are "destructive" or "dangerous" in the sense that they modify their receivers. The controversy arises because *not all* of the destructive methods are marked in this way. Shouldn't we be consistent?

No, in fact we should not. Some of the methods by their very nature change their receiver (such as the `Array` methods `replace` and `concat`). Some of them are "writer" methods, allowing assignment to a class attribute. We should *not* append an exclamation point to the attribute name or the equal sign. There are some methods that arguably change the state of the receiver, such as `read`; this occurs too frequently to be marked in this way. If every destructive method name ended in `!`, very soon our programs would look like sales brochures for a multilevel marketing firm.

Do you notice a kind of tension between opposing forces, a tendency for all rules to be violated? Let me state this as Fulton's Second Law: *Every rule has an exception, except Fulton's Second Law.*

What we see in Ruby is not a "foolish consistency" or a rigid adherence to a set of simple rules. In fact, perhaps part of the Ruby Way is that it is *not* a rigid and inflexible approach. In language design, as Matz once said, you should "follow your heart."

Yet another aspect of this philosophy is this: Do not fear change at runtime; do not fear what is dynamic. The world is dynamic; why should a programming language be static? Ruby is one of the more dynamic languages in existence.

I would also argue another aspect: Do not be a slave to performance issues. When performance is unacceptable, the issue must be addressed, but it should normally not be the first thing you think about. Prefer elegance over efficiency where efficiency is less than critical. Then again, if you are writing a library that may be used in unforeseen ways, performance may be critical from the start.

When I look at Ruby, I perceive a balance between different design goals—a complex interaction reminiscent of the *n*-body problem in physics. I can imagine it might be modeled as an Alexander Calder mobile. It is perhaps this interaction itself, the harmony, that embodies Ruby's philosophy rather than just the individual parts. Programmers know that their craft is not just science and technology, but art. I hesitate to say that there is a spiritual aspect to

computer science, but just between you and me, there certainly is. (If you have not read Robert Pirsig's *Zen and the Art of Motorcycle Maintenance*, I recommend that you do so.)

Ruby arose from the human urge to create things that are useful and beautiful. Programs written in Ruby should spring from that same God-given source. That, to me, is the essence of the Ruby Way.