Introduction

Graphical user interfaces (GUIs) are great for a lot of things; they are typically much kinder to newcomers than the stark glow of a cold, blinking cursor. This comes at a price: you can get only so proficient at a GUI before you have to learn its esoteric keyboard shortcuts. Even then, you will hit the limits of productivity and efficiency. GUIs are notoriously hard to script and automate, and when you can, your script tends not to be very portable.

This is all beside the point; we are software developers, and we write programs. What could be more natural than using code to get our work done? Consider the following command sequence:

```
> cd ~/Projects/cli
> vi chapter2.md
```

While these two commands might strike you as opaque, they are a highly efficient means of editing a file.

For most of my career, the command line meant a UNIX shell, like `bash`. The `bash` shell provides some basic built-in commands, as well as access to many other standard (and nonstandard) commands that are shipped with any UNIX system. These commands are single-purpose, require no user interaction, and come with easy-to-use (but hard-to-learn) user interfaces. These attributes let you piece them together in a near-infinite number of ways. Automating sophisticated behavior, performing complicated analysis, and parsing a myriad of text files can be done easily and expediently. This was life for me early on in my career. And it was good.

Then, in the mid-1990s, as Java grew in popularity, the idea of stringing together UNIX command-line utilities to get things done came to be seen as archaic. Java programs eschewed simple text-based configuration and file-based input/output (I/O) for complex hierarchies of XML driven by RPC and HTTP I/O. This allowed for very sophisticated systems to be built, and GUI tools sprang up to abstract away the complexity of building and configuring these systems. Even the act of writing and building code got swallowed up by ever more complex integrated development environments (IDEs). The simplicity of the command line was starting to get lost.

The problem is, there are too many tasks that don't fit the model of these tools; it's just too darn easy to go out to the shell and get things done. So, while I never bought into the concept that IDEs and sophisticated GUI tools were an advancement of the command line, I made peace with the facts of life and settled into a comfortable pattern: Java was for "real" code, and the command line (along with Perl and Ruby) was for automation, one-off scripts, and other things that helped me get repetitive things done quickly.

In the mid 2000s, I started to take notice of Ruby, Rails, and the amazing community built up around these tools. To my surprise (and delight), almost everything was command-line driven. Dynamic languages like Ruby don't lend themselves too well to IDEs (some even argue that an IDE makes no sense for such languages), and the burgeoning developer community wasn't on the radar of any top-tier tool makers. The community embraced the command line and created command-line applications for everything. Although Perl had been doing this for years, this was the first time I'd noticed such a strong embrace of the command line in the "post-Java" world.

What was more interesting was the taste and polish put into these command-line apps. Most featured a full-blown help system, often with command-based navigation of features, but still stayed true to the "UNIX way" of simplicity and interoperability. Take gem, for example. It's the command used to install other Ruby apps and libraries into your system:

```
$ gem help
RubyGems is a sophisticated package manager for Ruby.  This is a
basic help message containing pointers to more information.

  Usage:
    gem -h/--help
    gem -v/--version
    gem command [arguments...] [options...]

  Examples:
    gem install rake
    gem list --local
    gem build package.gemspec
    gem help install

  Further help:
    gem help commands            list all 'gem' commands
    gem help examples            show some examples of usage
    gem help platforms           show information about platforms
    gem help <COMMAND>          show help on COMMAND
                                  (e.g. 'gem help install')
    gem server                  present a web page at
                                http://localhost:8808/
                                with info about installed gems
  Further information:
    http://rubygems.rubyforge.org
```

This is just a small part of the very complete documentation available, and it's all there, right from the command line. It's clear that a lot of thought was put into making this tool polished; this was no one-off, hacky script. Much like the design philosophy of Ruby on Rails, there was clear care given to the user experience of the programmer. These tools aren't one-off scripts someone pieced together; they are made for "real" work.

What this told me was that the command line is far from the anachronism that Java tool vendors would have us believe; it's here to stay. The future of development won't just be manipulating buttons and toolbars and dragging and dropping icons to create code; the efficiency and productivity inherent to a command-line interface will always have a place in a good developer's tool chest. There are developers who demand polish and usability from their command-line tools, and there are developers who are interested in delivering it!

That's what this book is about: delivering awesome command-line applications (and how easy it is to do so in Ruby). It's for any programmer who wants to unlock the potential of a command-line interface but who also wants to create a polished and robust application with a real user interface that is easy to grasp and use.

How This Book Is Organized

In the next ten chapters, we'll discuss every detail of command-line application development, from user input, program output, and code organization to error handling, testing, and distribution. We'll learn about this by building and enhancing two example applications. Over the course of the book, we'll make them better and better to learn what an awesome command-line app is. We'll see that Ruby makes it very easy to do, thanks to its great syntax and features, as well as several open source libraries.

The first thing we'll learn—in Chapter 1, *Have a Clear and Concise Purpose*, on page 1—is what sort of applications are right for the command line. We'll then learn—in Chapter 2, *Be Easy to Use*, on page 13—the nuts and bolts of making an awesome application that's easy for both users and the system to interact with. That chapter is all about the user interface of command-line apps and introduces the two main styles of app: a simple UNIX-like style and the more complex "command-suite" style, as exemplified by commands like git or gem.

In Chapter 3, *Be Helpful*, on page 33, we'll learn how to provide excellent help and usage documentation; command-line apps are harder to discover and learn compared to GUIs, so this is one of the most important things to get right. We'll follow that up with Chapter 4, *Play Well with Others*, on page 53, where we'll learn how to make our apps interoperable with any other system.

At this point, we'll know how to make a *good* command-line app. Chapter 5, *Delight Casual Users*, on page 71 is where we take things to the next level and learn how easy it is to add polish to our apps. We'll continue this trend in Chapter 6, *Make Configuration Easy*, on page 89, where we'll learn how to make our apps easy to use for users with many different tastes and preferences.

Chapter 7, *Distribute Painlessly*, on page 101 will cover everything you need to distribute your application with RubyGems so that others can use it (we'll also cover installation in tightly controlled environments where RubyGems isn't an option).

In Chapter 8, *Test, Test, Test*, on page 117, we'll learn all about testing command-line apps, including some techniques to keep your tests from making a mess of your system. With the ability to test our apps comes the ability to refactor them so they are easier to maintain and enhance. Chapter 9, *Be Easy to Maintain*, on page 141 will cover some conventions around code organization, as well as some design patterns that are most useful to command-line apps.

We'll finish by pushing the envelope of what command-line apps should do in Chapter 10, *Add Color, Formatting, and Interactivity*, on page 153. We'll learn all about colored, formatted output, as well as interacting with the user using Readline.

Many open source libraries and tools help make command-line apps in Ruby. We'll look at some of them, such as OptionParser, GLI, and Cucumber, in great detail. But you don't have to limit yourself to just these tools. Appendix 1, *Common Command-Line Gems and Libraries*, on page 175 will go over many of the other popular libraries so you can use the best tool for you.

Who This Book Is For

This book is aimed at both developers and system administrators who have some familiarity with Ruby and who find themselves automating things on the command line (or who wish they could).

- If you're a developer who finds yourself faced with automation tasks but aren't familiar with the various conventions and techniques around the command line, this book will help you. A problem you might have is the maintenance of a "quick hack" script you wrote that has lived long past its prime. This book will give you the tools and techniques to make your next script longer-lived, polished, and bulletproof...all without spending a lot of time on it.

- If you're a sysadmin, you might find shell scripting limiting or frustrating. If you're pushing bash to the limit in your automation tasks, this book will open up a whole new world for you. Writing command-line apps in Ruby is also a great way to really learn Ruby and become a better programmer, since you can apply it directly to your day-to-day tasks.

What You'll Need

The only thing you'll need to follow along is a Ruby installation and a UNIX-like shell. You'll need to have either Ruby 2.0 or 2.1 for all the examples in this book to work, and the bash shell is recommended (although we aren't going to be seeing many shell-specific features). If you download the code from the book's website,[1] you'll notice at the top of the archive is a Gemfile. This should contain a list of all the gems you need to run the example apps, and you can use this file, along with Bundler,[2] to install everything in one step. If you don't know what any of that means, don't worry; the book will tell you when to install any needed gems. If things aren't working right, you can use the Gemfile to see which versions of gems I used when writing the book.

For writing command-line apps and following along with the examples, Mac and Linux users just need a text editor and a terminal or shell application (I'm assuming you'll have Ruby installed already; most Linux distributions include it). I highly recommend that you use RVM[3] and create a gemset for the examples in this book. RVM allows you to install any version of Ruby alongside your system version and to isolate gems from one another, which is very handy when learning new technologies.

For Windows users, the examples and code should work from the command prompt; however, you might have a better experience installing Cygwin[4] or

1. http://pragprog.com/book/dccar2
2. http://gembundler.com
3. https://rvm.io
4. http://www.cygwin.com/

MSYS[5] and using one of those for your shell. If you haven't installed Ruby, the easiest way to do that is to use the Ruby Installer.[6] For the most part, everything in this book is compatible with Windows, with the exception of the following:

- For apps with the suffix .rb, you will need to associate the file extension with Ruby. You should be able to do this when running the Ruby Installer. For apps that have no suffix, assuming you've set up the association to the .rb extension, you will need to run the app via the ruby command, like so:

  ```
  c:\> ruby my_app.rb
  ```

  To simplify things, you could create a .bat file to wrap this up:

  ```
  @echo off
  ruby my_app.rb %*
  ```

  The %* ensures that all the command-line parameters you give to your .bat will get passed along to your app.

- Aruba, the tool we'll be using to run acceptance tests of our command-line apps, is not well supported on Windows at the time of this writing. We'll cover this in more detail when we get to the chapter on testing, which is Chapter 8, *Test, Test, Test*, on page 117.

Other than that, if there's something a Windows user will need to do a bit differently, we'll point it out, but generally speaking, things work well on both UNIX-like platforms and Windows.

Conventions Used in the Book

There are three important things to know about the layout and conventions used in this book: the level of background knowledge you'll need on Ruby, UNIX, and OO; the way we'll work with code; and where testing fits into all this.

Ruby, UNIX, and Object Orientation

Since this is a book about writing command-line apps in Ruby, you're going to need to know a bit about the Ruby language and the UNIX environment. We've kept the code examples as clear as we can so that even with a passing familiarity with Ruby and UNIX, you'll be able to follow along.

Later in the book, we'll start to use more of the object-oriented features of Ruby, so knowing what classes and objects are will be helpful. Again, we've

5. http://www.mingw.org/wiki/MSYS
6. http://rubyinstaller.org/

kept it as simple as we could so you can focus on the tools and techniques without getting distracted by some of Ruby's more esoteric features.

If you're very new to Ruby or just want to brush up, please consider the Ruby Koans[7] and the "Pickaxe Book" (*Programming Ruby: The Pragmatic Programmer's Guide* [TFH13]).

Code

It's also worth pointing out that this book is about *code*. There is a lot of code, and we'll do our best to take each new bit of it step by step. Much of the code in this book will be from two example applications that we'll enhance and improve over time. To point out new things that we're changing, we'll use a subtle but important callout. Consider some Ruby code like so:

```
if !filename.nil?
  File.open(filename) do |file|
    file.readlines do |line|
      puts line.upcase
    end
  end
end
```

We might want to change that if to an unless to avoid the negative test.

```
➤ unless filename.nil?
    File.open(filename) do |file|
      file.readlines do |line|
        puts line.upcase
      end
    end
  end
```

Do you see the arrow next to the new unless statement? Look for those every time there's new code. Occasionally, we'll introduce a larger change to the code we're working on. In those cases, we'll call out particular lines for reference, like so:

```
❶ def upper_case_file(filename)
❷   unless filename.nil?
      File.open(filename) do |file|
        file.readlines do |line|
❸         puts line.upcase
        end
      end
    end
  end
```

7. http://rubykoans.com/

We can then discuss particular lines using a numbered list:

❶ Here we define a new method named upper_case_file.

❷ We check for nil here, so we don't get an exception from File.open.

❸ Finally, we uppercase the line we read from the file before printing it with puts.

Testing

The Ruby community loves testing; test-driven development is at the heart of many great Ruby applications, and the community has a wide variety of tools to make testing very easy. We'll even be looking at some in Chapter 8, *Test, Test, Test,* on page 117. We won't, however, be doing much testing until then. While you should absolutely test everything you do, it can be somewhat distracting to explain a concept or best practice in the context of a unit test, especially with some of the unique features and challenges of a command-line application.

So, don't take the lack of testing as an endorsement of cowboy coding.[8]. We're omitting the tests so you can take in the important parts of making an awesome command-line application. Once you're comfortable with these best practices, the information we'll discuss about testing will leave you with all the skills you need to test-drive your next command-line app.

Online Resources

At the website for this book,[9] you'll find the following:

- The full source code for all the sample programs used in this book.
- An errata page, listing any mistakes in the current edition (let's hope that will be empty!).
- A discussion forum where you can communicate directly with the author and other Ruby developers. You are free to use the source code in your own applications as you see fit.

Note: If you're reading the ebook, you can also click the little gray rectangle before the code listings to download that source file directly.

8. http://en.wikipedia.org/wiki/Cowboy_coding
9. http://pragprog.com/titles/dccar2

Acknowledgments

This book started as part of the Pragmatic Programmers' "PragProWriMo," which isn't much more than some budding authors posting their daily writing stats to a forum[10] every day during the month of November. This book is very different from the 170 pages I produced in November 2010, but I wrote almost every day, proving that I could actually produce a book's worth of material and that writing command-line applications in Ruby was a large enough topic to fill a book!

I had no particular plans to do anything with the manuscript I wrote, but when Travis Swicegood, author of *Pragmatic Version Control with Git* [Swi08], posted in the forum that his PragProWriMo manuscript had been accepted for development, I thought I'd submit mine as well. So, while Travis wasn't the inspiration for the material in this book, he certainly was the inspiration for turning this material *into* a book.

There are a lot of people to thank, but I have to start with my wife, Amy, who has been amazingly supportive and encouraging. She even let me install Ruby, vim, and Cygwin on her Windows laptop for testing.

I'd like to thank my editor, John Osborn, for his patience and advice as well as for inadvertently giving me a crash course in technical writing.

Next, I'd like to thank all the technical reviewers who gave me invaluable feedback on my manuscript at various stages of its development. They include Paul Barry, Daniel Bretoi, Trevor Burnham, Jeff Cohen, Ian Dees, Avdi Grimm, Wynn Netherland, Staffan Nöteberg, Noel Rappin, Eric Sendlebach, Christopher Sexton, and Matt Wynne.

Finally, I'd like to thank the many programmers who've contributed to the open source projects I mention in the book, including, but probably not limited to, the following: Aslak Hellesøy, TJ Holowaychuk, Ara Howard, Yehuda Katz, James Mead, William Morgan, Ryan Tomayko, Chris Wanstrath, and, of course Yukihiro "Matz" Matsumoto, who created such a wonderful language in which to write command-line apps.

With all that being said, let's get down to business and start making our command-line apps a lot more awesome!

10. http://forums.pragprog.com/forums/190