CHAPTER 1

Have a Clear and Concise Purpose

You need to solve a problem. It might be that you need two systems to talk to each other that weren't designed for it. Or you may need to run some automated yet complex task periodically. Or, you may want to build simple productivity tools to help you work. This is where the command line shines, and these are the kinds of problems you'll learn to solve in this book.

Although it may seem obvious that a focused, single-purpose app is more desirable than one with a "kitchen sink" full of features, it's especially important for command-line apps. The way in which command-line apps get input, are configured, and produce output is incredibly simple and, in some ways, limiting. As such, a system of many single-purpose apps is better than a system of fewer (or one) complex apps. Simple, single-purpose apps are easier to understand, are easier to learn, are easier to maintain, and lead to more flexible systems.

Think of your command-line tasks as a set of layers: with the basic foundation of the standard UNIX tools, you can create more complex but still focused command-line apps. Those can be used for even more complex apps, each built on simpler tools below. The popular version control system git follows this design: many of git's commands are "plumbing" and are not intended for regular use. These commands are then used to build "porcelain" commands, which are still simple and single-purpose but are built using the "plumbing." This design comes in handy because, every once in a while, you need to use the "plumbing" directly. You can do this because git was designed around tools that each have a clear and concise purpose.

This chapter will set the stage for everything we'll be learning in the book. We'll look at two common problems and introduce two command-line apps to solve them. As a means of demonstrating more clearly what we mean by having a "clear and concise purpose," each problem-solving app will get an

iteration in this chapter. The first version of each app will be naive and then quickly revised to be more single-purpose, so we can see firsthand the level of function we want our apps to have.

1.1 Problem 1: Backing Up Data

Suppose our small development team is starting work on our company's flagship web application. This application is heavily data-driven and highly complex, with many features and edge cases. To build it, we're going to use an Agile methodology, where we work in two-week "sprints." In each sprint, we'll have a list of "user stories" representing the work we're doing. To officially complete a user story, we'll need to demonstrate that story functioning properly in a shared development environment.

To be able to demonstrate working features, we'll have a set of databases with specially chosen data that can simulate all of our edge cases and user flows. Setting up this data is time-consuming because our app is complex, so even though this data is fake, we want to treat it like real production data and back it up. Since we're constantly changing the data as we work, we want to save the state of each database every single day of the current iteration. We also want to keep a backup of the state of each database at the end of every iteration. So, if we're on the fifth day of our third iteration, we want to be able to access a backup for iterations 1 and 2, as well as backups for the first four days of the third iteration.

Like with most teams, at our company, we can't rely on a system administrator to back it up for us; we're a fledgling start-up, and resources are limited. A command-line app to the rescue! We need an app that will do the following:

- Do a complete dump of any MySQL database
- Name the backup file based on the date of the backup
- Allow the creation of our "end-of-iteration" backup, using a different naming scheme
- Compress the backup files
- Delete backups from completed iterations

Let's take a quick stab at it. We'll set up a `Hash` that contains information about all the databases we want to back up, loop over it, and then use Ruby's backtick operator to call `mysqldump`, followed by `gzip`[1]. We'll also examine the first argument given to our app; if it's present, we'll take that to mean we

1. You need to have both of these commands installed. `gzip` is standard on most UNIX and Mac computers. `mysqldump` requires installing MySQL. You can learn about MySQL at http://dev.mysql.com/doc/refman/5.5/en/installing.html

want to do an "end-of-iteration" backup. Here's what our initial implementation looks like:

have-a-purpose/db_backup/bin/db_backup_initial.rb

```
#!/usr/bin/env ruby

databases = {
  big_client: {
    database: 'big_client',
    username: 'big',
    password: 'big',
  },
  small_client: {
    database: 'small_client',
    username: 'small',
    password: 'p@ssWord!',
  }
}

end_of_iter = ARGV.shift

databases.each do |name,config|
  if end_of_iter.nil?
    backup_file = config[:database] + '_' + Time.now.strftime('%Y%m%d')
  else
    backup_file = config[:database] + '_' + end_of_iter
  end
  mysqldump = "mysqldump -u#{config[:username]} -p#{config[:password]} " +
    "#{config[:database]}"

  `#{mysqldump} > #{backup_file}.sql`
  `gzip #{backup_file}.sql`
end
```

If you're wondering what's going on the very first line, see *Shebang: How the System Knows an App Is a Ruby Script*, on page 4. Notice how we use ARGV, which is an Array that Ruby sets with all the command-line arguments to detect whether this is an "end-of-iteration" backup. In that case, we assume that whatever the argument was should go into the filename, instead of the current date. We'd call it like so:

```
$ db_backup_initial.rb
# => creates big_client_20110103.sql.gz
# => creates small_client_20110103.sql.gz
$ db_backup_initial.rb iteration_3
# => creates big_client_iteration_3.sql.gz
# => creates small_client_iteration_3.sql.gz
```

Compiled programs include information in the executable file that tells that operating system how to start the program. Since programs written in a scripting language like Ruby don't need to be compiled, the operating system must have some other way to know how to run these types of apps. On UNIX systems, this is done via the first line of code, commonly referred to as the *shebang line*.[a]

The shebang line starts with a number sign (#), followed by an exclamation point (!), followed by the path to an interpreter that will be used to execute the program. This path must be an absolute path, and this requirement can cause problems on some systems. Suppose we have a simple app like so:

```
#!/usr/bin/ruby
puts "Hello World!"
```

For this app to work on any other system, there must be a Ruby interpreter located at */usr/bin/ruby*. This might not be where Ruby is installed, and for systems that use RVM (an increasingly high number do so), Ruby will *never* be available in */usr/bin*.

To solve this, the program */usr/bin/env*, which is much more likely to be installed at that location, can be used to provide a level of indirection. env takes an argument, which is the name of a command to run. It searches the path for this command and runs it. So, we can change our program to use a shebang like so:

```
#!/usr/bin/env ruby
puts "Hello world!"
```

This way, as long as Ruby is in our path somewhere, the app will run fine. Further, since the number sign is the comment character for Ruby, the shebang is ignored if you execute your app with Ruby directly: `ruby my_app.rb`.

a. http://en.wikipedia.org/wiki/Shebang_(Unix)

There are a lot of problems with this app and lots of room for improvement. The rest of the book will deal with these problems, but we're going to solve the biggest one right now. This app doesn't have a clear and concise purpose. It may appear to—after all, it is backing up and compressing our databases—but let's imagine a likely scenario: adding a third database to back up.

To support this, we'd need to edit the code, modify the `databases` Hash, and redeploy the app to the database server. We need to make this app simpler. What if it backed up only *one* database? If it worked that way, we would call the app one time for each database, and when adding a third database for backup, we'd simply call it a third time. No source code changes or redistribution needed.

To make this change, we'll get the database name, username, and password from the command line instead of an internal Hash, like this:

have_a_purpose/db_backup/bin/db_backup.rb

```
#!/usr/bin/env ruby
database = ARGV.shift
username = ARGV.shift
password = ARGV.shift
end_of_iter = ARGV.shift
if end_of_iter.nil?
  backup_file = database + "_" + Time.now.strftime("%Y%m%d")
else
  backup_file = database + "_" + end_of_iter
end
`mysqldump -u#{username} -p#{password} #{database} > #{backup_file}.sql`
`gzip #{backup_file}.sql`
```

Now, to perform our backup, we call it like so:

```
$ db_backup.rb big_client big big
# => creates big_client_20110103.sql.gz
$ db_backup.rb small_client small "p@ssWord!"
# => creates small_client_20110103.sql.gz
$ db_backup.rb big_client big big iteration_3
# => creates big_client_iteration_3.sql.gz
$ db_backup.rb medium_client medium "med_pass" iteration_4
# => creates medium_client_iteration_4.sql.gz
```

It may seem like we've complicated things, but our app is a lot simpler now and therefore easier to maintain, enhance, and understand. To set up our backups, we'd likely use cron (which is a UNIX tool for regularly scheduling things to be run) and have it run our app three times, once for each database.

We'll improve on db_backup.rb throughout the book, turning it into an awesome command-line app. Of course, automating specialized tasks is only one use of the command line. The command line can also be an excellent interface for simple productivity tools. As developers, we tend to be on the command line a lot, whether editing code, running a build, or testing new tools. Given that, it's nice to be able to manage our work without leaving the command line.

1.2 Problem 2: Managing Tasks

Most software development organizations use some sort of task management or trouble-ticket system. Tools like JIRA, Bugzilla, and Pivotal Tracker provide a wealth of features for managing the most complex workflows and tasks, all from your web browser. A common technique when programming is to take a large task and break it down into smaller tasks, possibly even breaking *those* tasks down. Suppose we're working on a new feature for our company's

flagship web application. We're going to add a Terms of Service page and need to modify the account sign-up page to require that the user accept the new terms of service.

In our company-wide task management tool, we might see a task like "Add Terms of Service Checkbox to Signup Page." That's the perfect level of granularity to track the work by our bosses and other interested stakeholders, but it's too coarse to drive our work. So, we'll make a task list of what needs to be done:

- Add new field to database for "accepted terms on date."
- Get DBA approval for new field.
- Add checkbox to HTML form.
- Add logic to make sure the box is checked before signing up is complete.
- Perform peer code review when all work is done.

Tracking such fine-grained and short-lived tasks in our web-based task manager is going to be too cumbersome. We could write this on a scrap of paper or a text file, but it would be better to have a simple tool to allow us to create, list, and complete tasks in order. That way, any time we come back to our computer, we can easily see how much progress we've made and what's next to do.

To keep things single-purpose, we'll create three command-line apps, each doing the one thing we need to manage tasks. todo-new.rb will let us add a new task, todo-list.rb will list our current tasks, and todo-done.rb will complete a task.

They will all work off a shared text file, named todo.txt in the current directory, and work like so:

```
$ todo-new.rb "Add new field to database for 'accepted terms on date'"
Task added
$ todo-new.rb "Get DBA approval for new field."
Task added
$ todo-list.rb
1 - Add new field to database for 'accepted terms on date'
    Created:   2011-06-03 13:45
2 - Get DBA approval for new field.
    Created:   2011-06-03 13:46
$ todo-done.rb 1
Task 1 completed
$ todo-list.rb
1 - Add new field to database for 'accepted terms on date'
    Created:   2011-06-03 13:45
    Completed: 2011-06-03 14:00
2 - Get DBA approval for new field.
    Created:   2011-06-03 13:46
```

We'll start with todo-new.rb, which will read in the task from the command line and append it to todo.txt, along with a timestamp.

have_a_purpose/todo/bin/todo-new.rb

```
#!/usr/bin/env ruby

new_task = ARGV.shift

File.open('todo.txt','a') do |file|
  file.puts "#{new_task},#{Time.now}"
  puts "Task added."
end
```

This is pretty straightforward; we're using a comma-separated-values format for the file that stores our tasks. todo-list.rb will now read that file, printing out what it finds and generating the ID number.

have_a_purpose/todo/bin/todo-list.rb

```
#!/usr/bin/env ruby

File.open('todo.txt','r') do |file|
  counter = 1
  file.readlines.each do |line|
    name,created,completed = line.chomp.split(/,/)
    printf("%3d - %s\n",counter,name)
    printf("      Created   : %s\n",created)
    unless completed.nil?
      printf("      Completed : %s\n",completed)
    end
    counter += 1
  end
end
```

Finally, for todo-done.rb, we'll read the file in and write it back out, stopping when we get the task the user wants to complete and including a timestamp for the completed date as well:

have_a_purpose/todo/bin/todo-done.rb

```
#!/usr/bin/env ruby

task_number = ARGV.shift.to_i

File.open('todo.txt','r') do |file|
  File.open('todo.txt.new','w') do |new_file|
    counter = 1
    file.readlines.each do |line|
      name,created,completed = line.chomp.split(/,/)
      if task_number == counter
        new_file.puts("#{name},#{created},#{Time.now}")
        puts "Task #{counter} completed"
```

```
      else
        new_file.puts("#{name},#{created},#{completed}")
      end

      counter += 1
    end
  end
end

`mv todo.txt.new todo.txt`
```

As with db_backup_initial.rb, this set of command-line apps has some problems. The most important, however, is that we've gone too far making apps clear and concise. We have three apps that share a lot of logic. Suppose we want to add a new field to our tasks. We'll have to make a similar change to all three apps to do it, and we'll have to take extra care to keep them in sync.

Let's turn this app into a *command suite*. A command suite is an app that provides a set of commands, each representing a different function of a related concept. In our case, we want an app named todo that has the clear and concise purpose of managing tasks but that does so through a command-style interface, like so:

```
$ todo new "Add new field to database for 'accepted terms on date'"
Task added
$ todo new "Get DBA approval for new field."
Task added
$ todo list
1 - Add new field to database for 'accepted terms on date'
    Created:   2011-06-03 13:45
2 - Get DBA approval for new field.
    Created:   2011-06-03 13:46
$ todo done 1
Task 1 completed
$ todo list
1 - Add new field to database for 'accepted terms on date'
    Created:   2011-06-03 13:45
    Completed: 2011-06-03 14:00
2 - Get DBA approval for new field.
    Created:   2011-06-03 13:46
```

The invocation syntax is almost identical, except that we can now keep all the code in one file. What we'll do is grab the first element of ARGV and treat that as the command. Using a case statement, we'll execute the proper code for the command. But, unlike the previous implementation, which used three files, because we're in one file, we can share some code, namely, the way in which we read and write our tasks to the file.

have_a_purpose/todo/bin/todo

```
#!/usr/bin/env ruby
TODO_FILE = 'todo.txt'

def read_todo(line)
  line.chomp.split(/,/)
end
def write_todo(file,name,created=Time.now,completed='')
  file.puts("#{name},#{created},#{completed}")
end
command = ARGV.shift
case command
when 'new'
  new_task = ARGV.shift
  File.open(TODO_FILE,'a') do |file|
    write_todo(file,new_task)
    puts "Task added."
  end
when 'list'
  File.open(TODO_FILE,'r') do |file|
    counter = 1
    file.readlines.each do |line|
      name,created,completed = read_todo(line)
      printf("%3d - %s\n",counter,name)
      printf("      Created   : %s\n",created)
      unless completed.nil?
        printf("      Completed : %s\n",completed)
      end
      counter += 1
    end
  end
when 'done'
  task_number = ARGV.shift.to_i
  File.open(TODO_FILE,'r') do |file|
    File.open("#{TODO_FILE}.new",'w') do |new_file|
      counter = 1
      file.readlines.each do |line|
        name,created,completed = read_todo(line)
        if task_number == counter
          write_todo(new_file,name,created,Time.now)
          puts "Task #{counter} completed"
        else
          write_todo(new_file,name,created,completed)
        end
        counter += 1
      end
    end
  end
  `mv #{TODO_FILE}.new #{TODO_FILE}`
end
```

Notice how the methods read_todo and write_todo encapsulate the format of tasks in our file? If we ever needed to change them, we can do it in just one place. We've also put the name of the file into a constant (TODO_FILE), so that can easily be changed as well.

1.3 What Makes an Awesome Command-Line App

Since the rest of this book is about what makes an awesome command-line app, it's worth seeing a broad overview of what we're talking about. In general, an awesome command-line app has the following characteristics:

Easy to use The command-line can be an unforgiving place to be, so the easier an app is to use, the better.

Helpful Being easy to use isn't enough; the user will need clear direction on *how* to use an app and how to fix things they might've done wrong.

Plays well with others The more an app can interoperate with other apps and systems, the more useful it will be, and the fewer special customizations that will be needed.

Has sensible defaults but is configurable Users appreciate apps that have a clear goal and opinion on how to do something. Apps that try to be all things to all people are confusing and difficult to master. Awesome apps, however, allow advanced users to tinker under the hood and use the app in ways not imagined by the author. Striking this balance is important.

Installs painlessly Apps that can be installed with one command, on any environment, are more likely to be used.

Fails gracefully Users will misuse apps, trying to make them do things they weren't designed to do, in environments where they were never designed to run. Awesome apps take this in stride and give useful error messages without being destructive. This is because they're developed with a comprehensive test suite.

Gets new features and bug fixes easily Awesome command-line apps aren't awesome just to use; they are awesome to hack on. An awesome app's internal structure is geared around quickly fixing bugs and easily adding new features.

Delights users Not all command-line apps have to output monochrome text. Color, formatting, and interactive input all have their place and can greatly contribute to the user experience of an awesome command-line app.

1.4 Moving On

The example apps we saw in this chapter don't have many aspects of an awesome command-line app. They're downright awful, in fact, but we have to start somewhere, and these are simple enough and general enough that we can demonstrate everything we need to know about making an awesome command-line app by enhancing them.

In this chapter, we learned the absolute most important thing for a command-line app: have a clear, concise purpose that solves a problem we have. Next, we'll learn how to make our app easier to use by implementing a more canonical command-line interface. As we work through the book, we'll make refinement after refinement, starting our focus on the general users of our app, then focusing on power users, and then worrying about other developers helping us with our app, before finally finishing with tools and techniques to help us maintain the app.

CHAPTER 2

Be Easy to Use

After installing your app, the first experience a user has with it will be the actual command-line interface. If the interface is difficult, counterintuitive, or, well, ugly, it's not going to inspire a lot of confidence, and your users will have a hard time using it to achieve its clear and concise purpose. Conversely, if it's easy to use, your interface will give your application an edge with its audience.

Fortunately, it's easy to get the command-line interface right, once you know the proper tools and techniques. The UNIX command line has a long and storied history, and there are now many conventions and idioms for how to invoke a command-line app. If your app follows these conventions, your users will have an easier time using it. We'll see that even a highly complex app can have a succinct and memorable interface.

In this chapter, we'll learn to use standard library and open source community tools that make it incredibly simple to create a conventional, idiomatic command-line interface whether it's a simple backup script or a complex command-line task management system. We'll learn how to make a simple command-line interface using Ruby's OptionParser class and then tackle a more sophisticated command-suite application, which we'll build using the open source GLI library. But first, we need to get familiar with the proper names of the elements of a typical command-line interface: its options, arguments, and commands.

2.1 Understanding the Command Line: Options, Arguments, and Commands

To tell a command-line application how to do its work, you typically need to enter more than just the name of its executable. For example, we must tell grep which files we want it to search. The database backup app, db_backup.rb, that we introduced in the previous chapter needs a username and password

and a database name in order to do its work. The primary way to give an app the information it needs is via *options* and *arguments*, as depicted in Figure 1, *Basic parts of a command-line app invocation*, on page 14. Note that this format isn't imposed by the operating system but is based on the GNU standard for command-line apps.[1] Before we learn how to make a command-line interface that can parse and accept options and arguments, we need to delve a bit deeper into their idioms and conventions. We'll start with options and move on to arguments. After that, we'll discuss *commands*, which are a distinguishing feature of command suites.

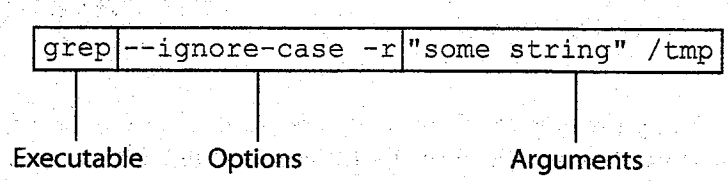


Figure 1—Basic parts of a command-line app invocation

Options

Options are the way in which a user modifies the behavior of your app. Consider the two invocations of ls shown here. In the first, we omit options and see the default behavior. In the second, we use the -l option to modify the listing format.

```
$ ls
one.jpg    two.jpg     three.jpg
$ ls -l
-rw-r--r--   1 davec  staff   14005 Jul 13 19:06 one.jpg
-rw-r--r--   1 davec  staff   14005 Jul 11 13:06 two.jpg
-rw-r--r--   1 davec  staff   14005 Jun 10 09:45 three.jpg
```

Options come in two forms: long and short.

Short-form options Short-form options are preceded by a dash and are only one character long, for example -l. Short-form options can be combined after a single dash, as in the following example. For example, the following two lines of code produce exactly the same result:

```
ls -l -a -t

ls -lat
```

1. http://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html

Long-form options Long-form options are preceded by two dashes and, strictly speaking, consist of two or more characters. However, long-form options are usually complete words (or even several words, separated by dashes). The reason for this is to be explicit about what the option means; with a short-form option, the single letter is often a mnemonic. With long-form options, the convention is to spell the word for what the option does. In the command `curl --basic http://www.google.com`, for example, `--basic` is a single, long-form option. Unlike short options, long options cannot be combined; each must be entered separately, separated by spaces on the command line.

Command-line options can be one of two types: *switches*, which are used to turn options on and off and do not take arguments, and *flags*, which take arguments, as shown in Figure 2, *A command-line invocation with switches and flags*, on page 15. Flags typically require arguments but, strictly speaking, don't need to do so. They just need to accept them. We'll talk more about this in Chapter 5, *Delight Casual Users*, on page 71.

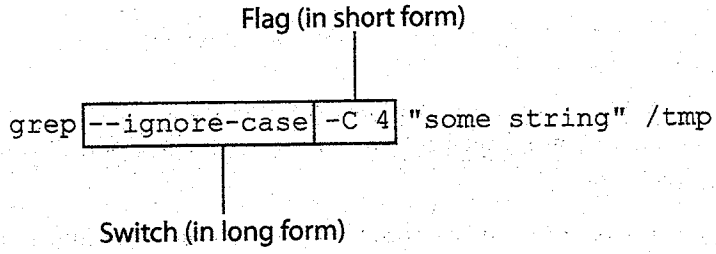


Figure 2—A command-line invocation with switches and flags

Typically, if a switch is in the long-form (for example `--foo`), which turns "on" some behavior, there is also another switch preceded with `no-` (for example `--no-foo`) that turns "off" the behavior.

Finally, long-form flags take their argument via an equal sign, whereas in the short form of a flag, an equal sign is typically not used. For example, the `curl` command, which makes HTTP requests, provides both short-form and long-form flags to specify an HTTP request method: `-X` and `--request`, respectively. The following example invocations show how to properly pass arguments to those flags:

```
curl -X POST http://www.google.com

curl --request=POST http://www.google.com
```

Although some apps do not require an equal sign between a long-form flag and its argument, your apps should always accept an equal sign, because this is the idiomatic way of giving a flag its argument. We'll see later in this chapter that the tools provided by Ruby and its open source ecosystem make it easy to ensure your app follows this convention.

Arguments

As shown in Figure 1, *Basic parts of a command-line app invocation*, on page 14, arguments are the elements of a command line that aren't options. Rather, arguments represent the objects that the command-line app will operate on. Typically, these objects are file or directory names, but this depends on the app. We might design our database backup app to treat the arguments as the names of the databases to back up.

Not all command-line apps take arguments, while others take an arbitrary number of them. Typically, if your app operates on a file, it's customary to accept any number of filenames as arguments and to operate on them one at a time.

Commands

Figure 1, *Basic parts of a command-line app invocation*, on page 14 shows a diagram of a basic command-line invocation with the main elements of the command line labeled.

For simple command-line applications, options and arguments are all you need to create an interface that users will find easy to use. Some apps, however, are a bit more complicated. Consider git, the popular distributed version control system. git packs a lot of functionality. It can add files to a repository, send them to a remote repository, examine a repository, or fetch changes from another user's repository. Originally, git was packaged as a collection of individual command-line apps. For example, to commit changes, you would execute the git-commit application. To fetch files from a remote repository, you would execute git-fetch. While each command provided its own options and arguments, there was some overlap.

For example, almost every git command provided a --no-pager option, which told git *not* to send output through a pager like more. Under the covers, there was a lot of shared code as well. Eventually, git was repackaged as a single executable that operated as a *command suite*. Instead of running git-commit, you run git commit. The single-purpose command-line app git-commit now becomes a *command* to the new command-suite app, git.

A command in a command-line invocation isn't like an option or an argument; it has a more specific meaning. A command is how you specify the action to take from among a potentially large or complex set of available actions. If you look around the Ruby ecosystem, you'll see that the use of command suites is quite common. gem, rails, and bundler are all types of command suites.

Figure 3, *Basic parts of a command-suite invocation*, on page 17 shows a command-suite invocation, with the command's position on the command line highlighted.

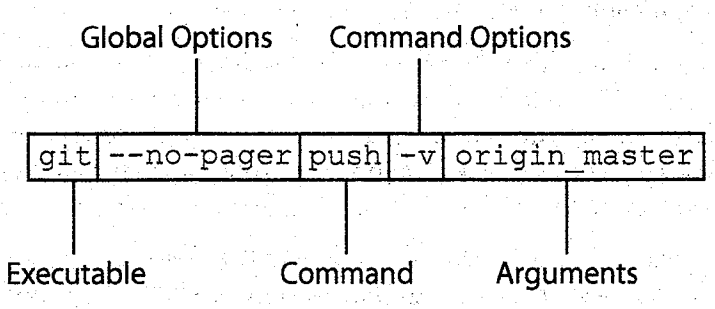


Figure 3—Basic parts of a command-suite invocation

You won't always design your app as a command suite; only if your app is complex enough that different behaviors are warranted will you use this style of interface. Further, if you *do* decide to design your app as a command suite, your app should *require* a command (we'll talk about how your app should behave when the command is omitted in Chapter 3, *Be Helpful*, on page 33).

The command names in your command suite should be short but expressive, with short forms available for commonly used or lengthier commands. For example, Subversion, the version control system used by many developers, accepts the short-form co in place of its checkout command.

A command suite can still accept options; however, their position on the command line affects how they are interpreted.

Global options Options that you enter before the command are known as *global options*. Global options affect the global behavior of an app and can be used with any command in the suite. Recall our discussion of the --no-pager option for git? This option affects all of git's commands. We know this because it comes before the command on the command line, as shown in Figure 3, *Basic parts of a command-suite invocation*, on page 17.

Command options Options that follow a command are known as *command-specific options* or simply command options. These options have meaning only in the context of their command. Note that they can also have the same names as global options. For example, if our to-do list app took a global option -f to indicate where to find the to-do list's file, the list command might also take an -f to indicate a "full" listing.

The command-line invocation would be `todo -f ~/my_todos.txt list -f`. Since the first -f comes before the command and is a global option, we won't confuse it for the second -f, which is a command option.

Most command-line apps follow the conventions we've just discussed. If your app follows them as well, users will have an easier time learning and using your app's interface. For example, if your app accepts long-form flags but doesn't allow the use of an equal sign to separate the flag from its argument, users will be frustrated.

The good news is that it's very easy to create a Ruby app that follows all of the conventions we've discussed in this section. We'll start by enhancing our Chapter 1 database backup app from Chapter 1, *Have a Clear and Concise Purpose*, on page 1 to demonstrate how to make an easy-to-use, conventional command-line application using OptionParser. After that, we'll use GLI to enhance our to-do list app, creating an idiomatic command suite that's easy for our users to use and easy for us to implement.

2.2 Building an Easy-to-Use Command-Line Interface

If you've done a lot of shell scripting (or even written a command-line tool in C), you're probably familiar with getopt,[2] which is a C library for parsing the command line and an obvious choice as a tool for creating your interface. Although Ruby includes a wrapper for getopt, you shouldn't use it, because there's a better built-in option: OptionParser. As you'll see, OptionParser is not only easy to use but is much more sophisticated than getopt and will result in a superior command-line interface for your app. OptionParser code is also easy to read and modify, making enhancements to your app simple to implement.

Before we see how to use OptionParser, let's first consider the input our application needs to do its job and the command line that will provide it. We'll use the backup application, db_backup.rb, which we introduced in Chapter 1, *Have a Clear and Concise Purpose*, on page 1. What kind of options might our application need?

2. http://en.wikipedia.org/wiki/Getopt

Right now, it needs the name of a database and some way of knowing when we're doing an "end-of-iteration" backup instead of a normal, daily backup. The app will also need a way to authenticate users of the database server we're backing up; this means a way for the user to provide a username and password.

Since our app will mostly be used for making daily backups, we'll make that its default behavior. This means we can provide a switch to perform an "end-of-iteration" backup. We'll use -i to name the switch, which provides a nice mnemonic (*i* for "iteration"). For the database user and password, -u and -p are obvious choices as flags for the username and password, respectively, as arguments.

To specify the database name, our app could use a flag, for example -d, but the database name actually makes more sense as an argument. The reason is that it really is the object that our backup app operates on. Let's look at a few examples of how users will use our app:

```
$ db_backup.rb small_client
# => does a daily backup of the "small_client" database

$ db_backup.rb -u davec -p P@55WorD medium_client
# => does a daily  backup of the "medium_client" database, using the
#    given username and password to login

$ db_backup.rb -i big_client
# => Do an "end of iteration" backup for the database "big_client"
```

Now that we know what we're aiming for, let's see how to build this interface with OptionParser.

Building a Command-Line Interface with OptionParser

To create a simple command-line interface with OptionParser, create an instance of the class and pass it a block. Inside that block, we create the elements of our interface using OptionParser methods. We'll use on to define each option in our command line.

The on itself takes a block, which is called when the user invokes the option it defines. For flags, the block is given the argument the user provided. The simplest thing to do in this block is to simply store the option used into a Hash, storing "true" for switches and the block argument for flags. Once the options are defined, use the parse! method of our instantiated OptionParser class to do the actual command-line parsing. Here's the code to implement the iteration switch and username and password flags of our database application:

be_easy_to_use/db_backup/bin/db_backup.rb

```
#!/usr/bin/env ruby

# Bring OptionParser into the namespace
require 'optparse'

options = {}
option_parser = OptionParser.new do |opts|

  # Create a switch
  opts.on("-i","--iteration") do
    options[:iteration] = true
  end

  # Create a flag
  opts.on("-u USER") do |user|
    options[:user] = user
  end

  opts.on("-p PASSWORD") do |password|
    options[:password] = password
  end

end

option_parser.parse!
puts options.inspect
```

As you can see by inspecting the code, each call to on maps to one of the command-line options we want our app to accept. What's not clear is how OptionParser knows which are switches and which are flags. There is great flexibility in the arguments to on, so the type of the argument, as well as its contents, controls how OptionParser will behave. For example, if a string is passed and it starts with a dash followed by one or more nonspace characters, it's treated as a switch. If there is a space and another string, it's treated as a flag. If multiple option names are given (as we do in the line opts.on("-i","--iteration")), then these two options mean the same thing.

Table 1, *Overview of OptionParser parameters to on*, on page 21 provides an overview of how a parameter to on will be interpreted; you can add as many parameters as you like, in any order. The complete documentation on how these parameters are interpreted is available on the rdoc for the make_switch method.[3]

3. http://ruby-doc.org/stdlib-2.0.0/libdoc/optparse/rdoc/OptionParser.html#method-i-make_switch

Effect	Example	Meaning
Short-form switch	-v	The switch -v is accepted on the command line. Any number of strings like this may appear in the parameter list and will all cause the given block to be called.
Long-form switch	--verbose	The switch --verbose is accepted. Any number of strings like this may appear in the parameter list and can be mixed and matched with the shorter form previously.
Negatable long-form switch	--[no-]verbose	Both --verbose and --no-verbose are accepted. If the no form is used, the block will be passed false; otherwise, true is passed.
Flag with required argument	-n NAME or --name NAME	The option is a *flag*, and it requires an argument. All other option strings provided as parameters will require flags as well (for example, if we added the string --username after the -u USER argument in our code, then --username would also require an argument; we don't need to repeat the USER in the second string). The value provided on the command line is passed to the block.
Flag with optional argument	-n [NAME] or --name [NAME]	The option is a flag whose argument is optional. If the flag's argument is omitted, the block will still be called, but nil will be passed.
Documentation	Any other string	This is a documentation string and will be part of the help output.

Table 1—Overview of OptionParser parameters to on

In the blocks given to on, our code simply sets a value in our options hash. Since it's just Ruby code, we can do more than that if we'd like. For example, we could sanity check the options and fail early if the argument to a particular flag were invalid.

Validating Arguments to Flags

Suppose we know that the usernames of all the database users in our systems are of the form first.last. To help our users, we can validate the value of the argument to -u before even connecting to the database. Since the block given to an on method call is invoked whenever a user enters the option it defines, we can check within the block for the presence of a period in the username value, as the following code illustrates:

be_easy_to_use/db_backup/bin/db_backup.rb

```
opts.on("-u USER") do |user|
  unless user =~ /^.+\..+$/
    raise ArgumentError,"USER must be in 'first.last' format"
  end
  options[:user] = user
end
```

Here, we raise an exception if the argument doesn't match our regular expression; this will cause the entire option-parsing process to stop, and our app will exit with the error message we passed to raise.

You can probably imagine that in a complex command-line app, you might end up with a lot of argument validation. Even though it's only a few lines of extra code, it can start to add up. Fortunately, OptionParser is far more flexible than what we've seen so far. The on method is quite sophisticated and can provide a lot of validations for us. For example, we could replace the code we just wrote with the following to achieve the same result:

be_easy_to_use/db_backup/bin/db_backup.rb

```
opts.on("-u USER",
        /^.+\..+$/) do |user|
  options[:user] = user
end
```

The presence of a regular expression as an argument to on indicates to OptionParser that it should validate the user-provided argument against this regular expression. Also note that if you include any capturing groups in your regexp (by using parentheses to delineate sections of the regexp), those values will be extracted and passed to the block as an Array. The raw value from the command line will be at index 0, and the extracted values will fill out the rest of the array.

You don't have to use regular expressions for validation, however. By including an Array in the argument list to on, you can indicate the complete list of acceptable values. By using a Hash, OptionParser will use the keys as the acceptable values and send the mapped value to the block, like so:

```
servers = { 'dev' => '127.0.0.1',
            'qa' => 'qa001.example.com',
            'prod' => 'www.example.com' }

opts.on('--server SERVER',servers) do |address|
  # for --server=dev, address would be '127.0.0.1'
  # for --server=prod, address would be 'www.example.com'
end
```

Finally, if you provide a classname in the argument list, OptionParser will attempt to convert the string from the command line into an instance of the given class. For example, if you include the constant Integer in the argument list to on, OptionParser will attempt to parse the flag's argument into an Integer instance for you. There is support for many conversions. See *Type Conversions in OptionParser*, on page 24 for the others available and how to make your own using the accept method.

By using OptionParser, we've written very little code but created an idiomatic UNIX-style interface that will be familiar to anyone using our app. We've seen how to use this to improve our backup app, but how can we create a similarly idiomatic interface for our to-do list app? Our to-do list app is actually a series of commands: "create a new task," "list the tasks," "complete a task." This sounds like a job for the command-suite pattern.

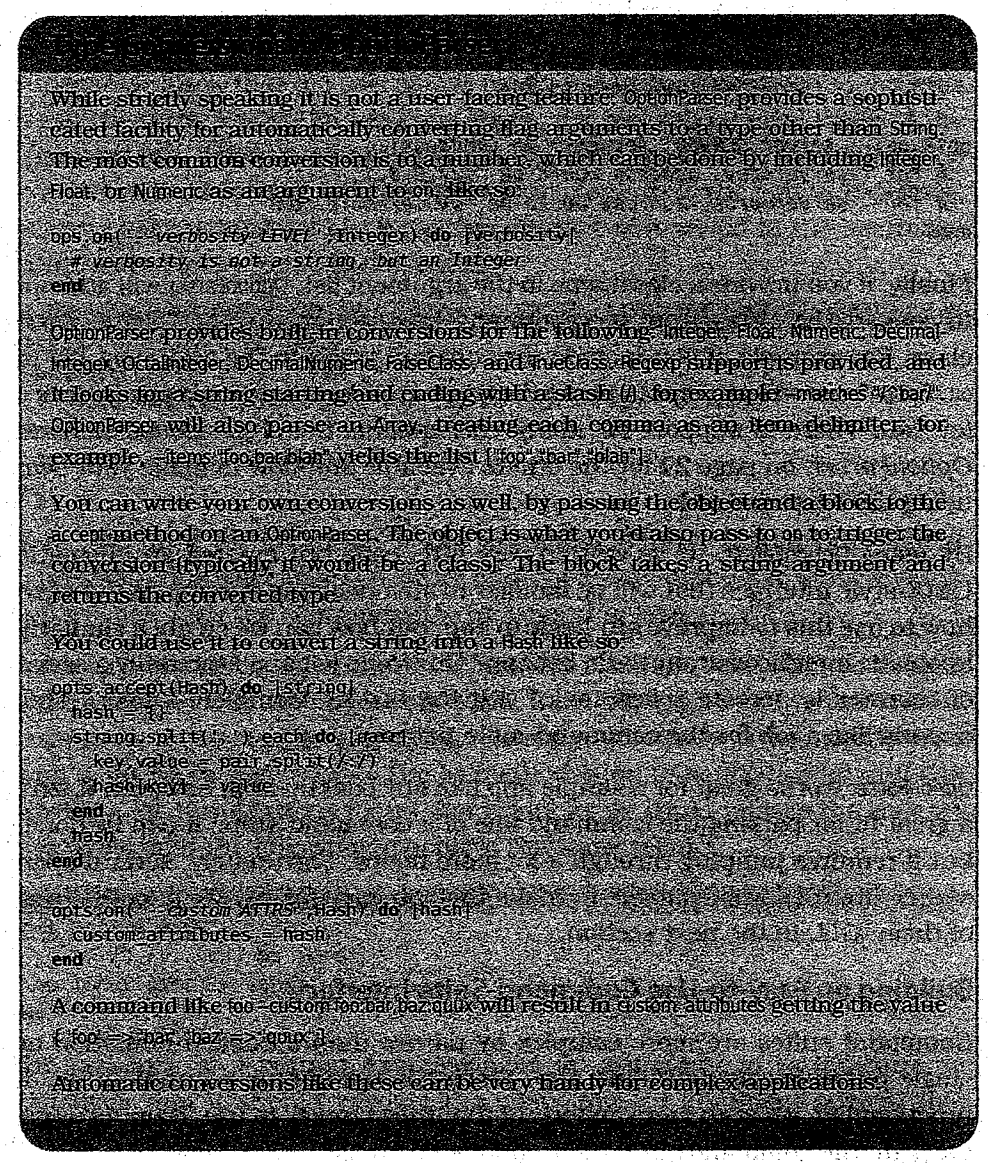
OptionParser works great for a simple app like our backup app; however, it isn't a great fit for parsing the command line of a command suite; it can be done, but it requires jumping through a lot more hoops. Fortunately, several open source libraries are available to make this job easy for us. We'll look at one of them, GLI, in the next section.

2.3 Building an Easy-to-Use Command-Suite Interface

Command suites are more complex by nature than a basic automation or single-purpose command-line app. Since command suites bundle a lot of functionality, it's even more important that they be easy to use. Helping users navigate the commands and their options is crucial.

Let's revisit our to-do list app we discussed in Chapter 1, *Have a Clear and Concise Purpose*, on page 1. We've discussed that the command-suite pattern is the best approach, and we have already identified three commands the app will need: "new," "list," and "done" to create a new task, list the existing tasks, and complete a task, respectively.

We also want our app to provide a way to locate the to-do list file we're operating on. A global option named -f would work well (*f* being a mnemonic for



While strictly speaking it is not a user-facing feature, OptionParser provides a sophisticated facility for automatically converting flag arguments to a type other than String. The most common conversion is to a number, which can be done by including Integer, Float, or Numeric as an argument to on, like so:

```
opts.on("--verbosity LEVEL", Integer) do |verbosity|
  # verbosity is not a string, but an Integer
end
```

OptionParser provides built-in conversions for the following: Integer, Float, Numeric, Decimal-Integer, OctalInteger, DecimalNumeric, FalseClass, and TrueClass. Regexp support is provided, and it looks for a string starting and ending with a slash (/), for example --matches "/bar/". OptionParser will also parse an Array, treating each comma as an item delimiter, for example, --items "foo,bar,baz" yields the list ["foo","bar","baz"].

You can write your own conversions as well, by passing the object and a block to the accept method on an OptionParser. The object is what you'd also pass to on to trigger the conversion (typically it would be a class). The block takes a string argument and returns the converted type.

You could use it to convert a string into a Hash like so:

```
opts.accept(Hash, /\w+=\w+/) do |string|
  hash = {}
  string.split(",").each do |pair|
    key,value = pair.split(/=/)
    hash[key] = value
  end
  hash
end

opts.on("--custom ATTRS", Hash) do |hash|
  custom_attributes = hash
end
```

A command like foo --custom foo=bar,baz=qux will result in custom_attributes getting the value {"foo"=>"bar","baz"=>"qux"}.

Automatic conversions like these can be very handy for complex applications.

"file"). It would be handy if our "new" command allowed us to set a priority or place a new task directly at the top of our list. -p is a good name for a flag that accepts a priority as an argument, and we'll use -f to name a switch that means "first in the list."

We'll allow our list command to take a sort option, so it will need a flag named -s. done won't need any special flags right now. Let's see a few examples of the interface we want to create:

```
$ todo new "Rake leaves"
# => Creates a new todo in the default location

$ todo -f /home/davec/work.txt new "Refactor database"
# => Creates a new todo in /home/davec/work.txt instead
#    of the default

$ todo -f /home/davec/work.txt new "Do design review" -f
# => Create the task "Do design review" as the first
#    task in our task list in /home/davec/work.txt

$ todo list -s name
# => List all of our todos, sorted by name

$ todo done 3
# => Complete task #3
```

Unfortunately, OptionParser was not built with command suites in mind, and we can't directly use it to create this sort of interface. To understand why, look at our third invocation of the new command: both the "filename" global flag and the command-specific "first" switch have the same name: -f. If we ask OptionParser to parse that command line, we won't be able to tell which -f is which.

A command-line interface like this is too complex to do "by hand." What we need is a tool custom-built for parsing the command line of a command suite.

Building a Command Suite with GLI

Fortunately, many open source tools are available to help us parse the command-suite interface we've designed for our to-do list app. Three common ones are commander,[4] thor,[5] and GLI.[6] They are all quite capable, but we're going to use GLI here. GLI is actively maintained, has extensive documentation, and was special-built for making command-suite apps very easily (not to mention written by the author of this book). Its syntax is similar to commander and thor, with all three being inspired by rake; therefore, much of what we'll learn here is applicable to the other libraries (we'll see how to use them in a bit more depth in Appendix 1, *Common Command-Line Gems and Libraries*, on page 175).

Rather than modify our existing app with GLI library calls, we'll take advantage of a feature of GLI called *scaffolding*. We'll use it to bootstrap our app's UI and show us immediately how to declare our user interface.

4. http://visionmedia.github.com/commander/
5. https://github.com/wycats/thor
6. https://github.com/davetron5000/gli

Building a Skeleton App with GLI's scaffold

Once we install GLI, we can use it to bootstrap our app. The gli application is itself a command suite, and we'll use the scaffold command to get started. gli scaffold takes an arbitrary number of arguments, each representing a command for our new command suite. You don't have to think of all your commands up front. Adding them later is simple, but for now, as the following console session shows, it's easy to set up the commands you know you will need. For our to-do app, these include new, list, and done.

```
$ gem install gli
Successfully installed gli-2.8.0
1 gem installed
$ gli scaffold todo new list done
Creating dir ./todo/lib...
Creating dir ./todo/bin...
Creating dir ./todo/test...
Created ./todo/bin/todo
Created ./todo/README.rdoc
Created ./todo/todo.rdoc
Created ./todo/todo.gemspec
Created ./todo/test/default_test.rb
Created ./todo/test/test_helper.rb
Created ./todo/Rakefile
Created ./todo/Gemfile
Created ./todo/features
Created ./todo/lib/todo/version.rb
Created ./todo/lib/todo.rb
$ cd todo
$ bundle install
```

Don't worry about all those files that scaffold creates just yet; we'll explain them in future chapters. Now, let's test the new interface before we look more closely at the code:

```
$ bundle exec bin/todo new
$ bundle exec bin/todo done
$ bundle exec bin/todo list
$ bundle exec bin/todo foo
error: Unknown command 'foo'. Use 'todo help' for a list of commands
```

As you can see from the session dialog, our scaffolded app recognizes our commands, even though they're not yet implemented. We even get an error when we try to use the command foo, which we didn't declare. Don't worry about bundle exec; we'll explain the usage in future chapters.

Let's now look at the code GLI produces to see how it works. As you can see, GLI generated only the code it needs to parse the commands we passed as

arguments to the scaffold command. The switches and flags set by GLI are provided here as examples. We'll cover how to customize them later.

We'll go through the generated code step by step. First, we need to set up our app to bring GLI's libraries in, via a require and an include.

be_easy_to_use/todo/bin/todo

```
#!/usr/bin/env ruby

require 'gli'
include GLI::App
```

Since we've included GLI, the remaining code is mostly method calls from the GLI module.[7] The next thing the code does is to declare some global options.

be_easy_to_use/todo/bin/todo

```
switch  :s
flag    [:f,:filename]
```

This declares that the app accepts a global switch -s and a global flag -f. Remember, these are just examples; we'll change them later to meet our app's requirements. Next, the code defines the new command:

be_easy_to_use/todo/bin/todo

```
command :new do |c|
  c.switch :s
  c.flag   :f
  c.action do |global_options,options,args|
    # Your command logic here

    # If you have any errors, just raise them
    # raise "that command made no sense"
  end
end
```

The block given to command establishes a context to declare command-specific options via the argument passed to the block (c). GLI has provided an example of command-specific options by declaring that the new command accepts a switch -s and a flag -f. Finally, we call the action method on c and give it a block. This block will be executed when the user executes the new command and is where we'd put the code to implement new. The block will be given the parsed global options, the parsed command-specific options, and the command-line arguments via global_options, options, and args, respectively.

GLI has generated similar code for the other commands we specified to gli scaffold:

7. http://davetron5000.github.io/gli/rdoc/classes/GLI.html

be_easy_to_use/todo/bin/todo
```
command :list do |c|
  c.action do |global_options,options,args|
  end
end
command :done do |c|
  c.action do |global_options,options,args|
  end
end
```

The last step is to ask GLI to parse the command line and run our app. The run method returns with an appropriate exit code for our app (we'll learn all about exit codes in Chapter 4, *Play Well with Others*, on page 53).

be_easy_to_use/todo/bin/todo
```
exit run(ARGV)
```

GLI has provided us with a skeleton app that parses the command line for us; all we have to do is fill in the code (and replace GLI's example options with our own).

Turning the Scaffold into an App

As we discussed previously, we need a global way to specify the location of the to-do list file, and we need our new command to take a flag to specify the position of a new task, as well as a switch to specify "this task should go first." The list command needs a flag to control the way tasks are sorted.

Here's the GLI code to make this interface. We've also added some simple debugging, so when we run our app, we can see that the command line is properly parsed.

be_easy_to_use/todo/bin/todo_integrated.rb
```
flag :f
command :new do |c|
  c.flag :priority
  c.switch :f

  c.action do |global_options,options,args|
    puts "Global:"
    puts "-f - #{global_options[:f]}"
    puts "Command:"
    puts "-f - #{options[:f] ? 'true' : 'false'}"
    puts "--priority - #{options[:priority]}"
    puts "args - #{args.join(',')}"
  end
end
command :list do |c|
  c.flag :s
  c.action do |global_options,options,args|
```

```
    puts "Global:"
    puts "-f - #{global_options[:f]}"
    puts "Command:"
    puts "-s - #{options[:s]}"
  end
end
command :done do |c|
  c.action do |global_options,options,args|
    puts "Global:"
    puts "-f - #{global_options[:f]}"
  end
end
```

The highlighted code represents the changes we made to what GLI generated. We've removed the example global and command-specific options and replaced them with our own. Note that we can use both short-form and long-form options; GLI knows that a single-character symbol like :f is a short-form option but a multicharacter symbol like :priority is a long-form option.

We also added some calls to puts that demonstrate how we access the parsed command line (in lieu of the actual logic of our to-do list app). Let's see it in action:

```
$ bundle exec bin/todo -f ~/todo.txt new -f "A new task" "Another task"
Global:
-f - /Users/davec/todo.txt
Command:
-f - true
-p -
args - A new task,Another task
```

We can see that :f in global_options contains the file specified on the command line; that options[:f] is true, because we used the command-specific option -f; and that options[:priority] is missing, since we didn't specify that on the command line at all.

Once we've done this, we can add our business logic to each of the c.action blocks, using global_options, options, and args as appropriate. For example, here's how we might implement the logic for the to-do app list command:

```
c.action do |global_options,options,args|
  todos = read_todos(global_options[:filename])
  if options[:s] == 'name'
    todos = todos.sort { |a,b| a <=> b }
  end
  todos.each do |todo|
    puts todo
  end
end
```

We've used very few lines of code yet can parse a sophisticated user interface. It's a UI that users will find familiar, based on their past experience with other command suites. It also means that when we add more features to our app, it'll be very simple.

Is there anything else that would be helpful to the user on the command line? Other than some help documentation (which we'll develop in the next chapter), it would be nice if users could use the tab-completion features of their shell to help complete the commands of our command suite. Although our to-do app has only three commands now, it might need more later, and tab completion is a big command-line usability win.

Adding Tab Completion with GLI help and bash

An advantage of defining our command-suite's user interface in the declarative style supported by GLI is that the result provides us with a model of our UI that we can use to do more than simply parse the command line.

We can use this model, along with the sophisticated completion function of bash, to let the user tab-complete our suite's commands. First we tell bash that we want special completion for our app, by adding this to our ~/.bashrc and restarting our shell session:

```
complete -F get_todo_commands todo
```

The complete command tells bash to run a function (in our case, get_todo_commands) whenever a user types the command (in our case, todo) followed by a space and some text (optionally) and then hits the a Tab key (i.e., is asked to complete something). complete expects the function to return the possible matches in the shell variable COMPREPLY, as shown in the implementation of get_todo_commands (which also goes in our .bashrc):

```
function get_todo_commands()
{
    if [ -z $2 ] ; then
        COMPREPLY=(`todo help -c`)
    else
        COMPREPLY=(`todo help -c $2`)
    fi
}
```

Every GLI-powered app includes a built-in command called help that is mostly used for getting online help (we'll see more about this in the next chapter). This command also takes a switch and an optional argument you can use to facilitate tab completion.

The switch -c tells help to output the app's commands in a format suitable for bash completion. If the argument is also provided, the app will list only those commands that match the argument. Since our bash function is given an optional second argument representing what the user has entered thus far on the command line, we can use that to pass to help.

The end result is that your users can use tab completion with your app, and the chance of entering a nonexistent command is now very minimal—all without having to lift a finger! Note that for this to work, you must have todo installed in your PATH (we'll see how users can do this in Chapter 7, *Distribute Painlessly*, on page 101).

```
$ todo help -c
done
help
list
new
$ todo <TAB>
done help list new
$ todo d<TAB>
$ todo done
```

2.4 Moving On

We've learned in this chapter how simple it is to make an easy-to-use interface for a command-line application using built-in or open source libraries. With tools like OptionParser and GLI, you can spend more time on your app and rest easy knowing your user interface will be top notch and highly usable, even as you add new and more complex features.

Now that we know how to easily design and parse a good command-line interface, we need to find a way to let the user know how it works. In the next chapter, we'll talk about in-app help, specifically how OptionParser and GLI make it easy to create and format help text, as well as some slightly philosophical points about what makes good command-line help.

CHAPTER 3

Be Helpful

In the previous chapter, we learned how to make an easy-to-use command-line interface. We learned the elements that make a well-formed command-line interface and how to design simple apps and command suites that accept arguments, flags, switches, and commands in an unsurprising[1] way. What we didn't talk about was how a user finds out what options and commands such apps provide, what their options mean, and what arguments they accept or require. Without this information, our app might do the job expected of it, but it won't be very helpful.

Fortunately for us, the standard Ruby library OptionParser and the open source GLI gem give us the power to make our app helpful without a lot of effort. In fact, you'll see that it's actually *harder* to make an unhelpful app using these tools. We'll begin by exploring how you can add help and documentation to the pair of apps—db_backup.rb and todo—whose UI we developed in the previous chapter. We'll also look at ways to create more detailed user documentation with an open source library that can bundle UNIX-style manual pages with our app. We'll end the chapter with a look at some rules of thumb for making our documentation useful to both new users of our software and seasoned veterans.

3.1 Documenting a Command-Line Interface

An experienced command-line user will try one or two things on the command line to discover how to use an app: they will run it without arguments or give it a help switch, such as -h or --help (-help is also a possibility because many X-Windows apps respond to this for help). In each case, the user will expect to see a one-screen summary of the app's usage, including what arguments the app accepts or requires and what options are available.

1. http://en.wikipedia.org/wiki/Principle_of_least_astonishment

Because db_backup.rb uses OptionParser, we're most of the way there already. Apps that use OptionParser respond to -h and --help in just the way our users expect. When OptionParser encounters either of these switches on the command line (assuming you haven't overridden them), it will display basic help text that shows how to invoke the app and what options it accepts. Here's what OptionParser displays when a user enters an -h or --help option for db_backup.rb:

```
$ db_backup.rb -h
Usage: db_backup [options]
    -i, --iteration
    -u USER
    -p PASSWORD
$ db_backup.rb --help
Usage: db_backup [options]
    -i, --iteration
    -u USER
    -p PASSWORD
```

While OptionParser nicely formats the help screen for us, what's still missing is documentation to explain the meaning of each option. Even though the flags are somewhat self-documenting (e.g., a user will likely figure out that "PASSWORD" is the database password), they still bear further explanation. For example, because usernames are required to be in a certain format, the app should let users know that. The app also requires an argument—the name of the database to back up—and this should be documented in the help text as well.

Documenting Command-Line Options

Once we fill in the documentation, we'd like our help text to look like so:

```
$ db_backup.rb --help
Usage: db_backup [options]
    -i, --iteration   Indicate that this backup is an "iteration" backup
    -u USER           Database username, in first.last format
    -p PASSWORD       Database password
```

Now the user can see exactly what the options mean and what constraints are placed on them (e.g., the username's format). Achieving this with OptionParser couldn't be simpler. If you recall from Table 1, *Overview of OptionParser parameters to on*, on page 21, any string given as a parameter to on that doesn't match the format of an option will be treated as documentation.

So, all we need to do is add some strings to the end of our argument list to each of calls to on:

be_helpful/db_backup/bin/db_backup.rb

```
opts.on('-i','--iteration',
➤       'Indicate that this backup is an "iteration" backup') do
  options[:iteration] = true
end
opts.on('-u USER',
➤       'Database username, in first.last format',
        /^[^.]+\.[^.]+$/) do |user|
  options[:user] = user
end

opts.on('-p PASSWORD',
➤       'Database password') do |password|
  options[:password] = password
end
```

That's all there is to it—not bad for about thirty seconds of coding! Next, we need to document that our app takes the name of the database to back up as an argument.

Documenting Command-Line Arguments

OptionParser provides no way to explicitly document the arguments that a command-line app accepts or requires. You'll note that OptionParser does, however, display an invocation template as its first line of help text (Usage: db_backup.rb [options]). This is called the *banner* and is the perfect place to document our app's arguments. We'd like to append a description of our app's argument to OptionParser's banner so that our help screen looks like so:

```
$ bin/db_backup.rb -h
➤ Usage: db_backup.rb [options] database_name

    -i, --iteration   Indicate that this backup is an "iteration" backup
    -u USER           Database username, in first.last format
    -p PASSWORD       Database password
```

Did you notice that the string database_name now appears in the highlighted line? This is just enough information to tell the user that we require an argument and that it should be the name of the database. OptionParser has a property, banner, that we can set to accomplish this. Since our app currently doesn't set the banner, we get the default that we saw previously. Unfortunately, we cannot directly access this string and tack on database_name, so we'll have to re-create it ourselves.

The other tricky bit is that we don't want to hard-code the name of our app in the banner. If we did, we'd have to update our documentation if we chose to rename our app.

Fortunately, Ruby provides an answer. When an app runs, Ruby sets the global variable $PROGRAM_NAME to the full path name of the app's *executable*, which is the name of the physical file on disk that the operating system uses to run our app. The filename (without the full path) is the name of our app and what the user will type on the command line to run it, so we want to show only that.

Ruby's File class has a handy method named basename that will give us just the name of the file of our executable, without the path to it, which is exactly what we need to create our banner.

be_helpful/db_backup/bin/db_backup.rb

```
option_parser = OptionParser.new do |opts|
  executable_name = File.basename($PROGRAM_NAME)
  opts.banner = "Usage: #{executable_name} [options] database_name"
```

Now the user can easily see that our app requires one argument: the name of the database to back up. Note that we are using an underscore notation here; if we had written "database name" instead (using a space between the two words), a user might misinterpret the words as calling for two arguments, one called "database" and another called "name."

It's hard to think of adding one string to our app's help text as "documentation," but for apps as straightforward as ours, this is sufficient. The user knows that db_backup.rb backs up a database, and the string database_name is all the user needs in order to know that our argument is the name of the database to back up.

Some apps have more complex arguments, and we'll see later how we can bundle more detailed documentation with our app to explain them.

The last thing we need to do is to provide a brief summary of the purpose of our app so that occasional users can get a quick reminder of what it does.

Adding a Brief Description for a Command-Line Application

A user who has just installed our app will certainly remember its purpose, but someone running it weeks or months from now might not. Although it's not hard to guess that an app named db_backup backs up a database, occasional users might not recall that it's only for backing up MySQL databases and won't work on, say, an Oracle database. To be helpful to these users, db_backup.rb --help should include a brief summary of the app's purpose. This should be the first thing the user sees when asking for help, like so:

```
$ bin/db_backup.rb -h
➤ Backup one or more MySQL databases

Usage: db_backup.rb [options] database_name

    -i, --iteration   Indicate that this backup is an "iteration" backup
    -u USER           Database username, in first.last format
    -p PASSWORD       Database password
```

Like the usage statement, OptionParser doesn't provide a place to explicitly document our app's purpose, but we can add it to the banner, just like we did when we documented its arguments. Since the banner is going to be multiline, we can format it directly in our source using multiple lines (instead of putting control characters like \n in a single-line string) so that the banner text is easy to read and modify:

be_helpful/db_backup/bin/db_backup.rb

```
option_parser = OptionParser.new do |opts|
  executable_name = File.basename($PROGRAM_NAME)
  opts.banner = "Backup one or more MySQL databases

Usage: #{executable_name} [options] database_name

"
```

You might be tempted to add more documentation to the banner, but this is not what the banner is for. The banner should be brief and to the point, designed as reference. We'll see later in this chapter how we can provide more detailed help and examples.

Now that we've fully documented what our app does, how to invoke it, and what options are available, db_backup.rb seems pretty darn helpful. There's only one thing left to consider: what if the user executes db_backup.rb but omits the required argument, the database name?

We mentioned earlier that experienced command-line users might do this on purpose, as a way to get a help statement. The user could also do this by accident, forgetting to provide a database name. No matter what the user's intent might be, our app behaves the same: unhelpfully. It will likely generate an exception or, worse, fail silently.

In cases like this, where you don't know whether the user made a mistake or is just looking for help, you should cover both bases and provide an error message, followed by the help text. Let's see how to do this by looking at db_backup.rb.

Ruby places all command-line arguments in an array called ARGV, which OptionParser modifies when parse! is called. OptionParser's modification to ARGV is to remove all the options and arguments it knows about. What's left in ARGV are the unparsed arguments, which you can safely treat as the arguments the user provided on the command line. Unrecognized *switches and flags* will cause OptionParser to print an error and exit your app, so you'll never find them in ARGV.

All we need to do to detect this "request for help or erroneous invocation" situation is check that ARGV is empty *after* having OptionParser parse the command line, as shown in the following code:

be_helpful/db_backup/bin/db_backup.rb

```
option_parser.parse!
➤ if ARGV.empty?
    puts "error: you must supply a database_name"
    puts
    puts option_parser.help
  else
    database_name = ARGV[0]
    # proceed as normal to backup database_name
  end
```

Now db_backup.rb is as helpful as it can be:

```
$ db_backup.rb
➤ error: you must supply a database name

Backup one or more MySQL databases

Usage: db_backup.rb [options] database_name

    -i, --iteration  Indicate that this backup is an "iteration" backup
    -u USER          Database username, in first.last format
    -p PASSWORD      Database password
```

We've seen how easy it is to make a helpful user interface for simple command-line apps using OptionParser, but what about command suites? It's doubly important to provide a helpful user interface, because a command suite is naturally more complex. In the next section, we'll see how to do that by enhancing our to-do list app todo.

3.2 Documenting a Command Suite

Since command suites like todo are more complex than simpler command-line apps like db_backup.rb, it's important that we have documentation and that it's easy to access. Users need to know not only what each option does and what the arguments mean but also what commands are available and what they

do. The best way to provide this information is via a two-level help system. At the top "level," we see the "banner"-type information, the global options, the list of commands, and what each command does. This information should be provided when the app is invoked with no arguments or when invoked with the command `help`, like so:

```
$ bin/todo help
NAME
    todo -

SYNOPSIS
    todo [global options] command [command options] [arguments...]

GLOBAL OPTIONS
    -f, --filename=todo_file - Path to the todo file (default:
                               /Users/davec/.todo.txt)
    --help                   - Show this message

COMMANDS
    done - Complete a task
    help - Shows a list of commands or help for one command
    list - List tasks
    new  - Create a new task in the task list
```

The second "level" is where help on a particular command is displayed. This type of help can include more detail about what the command does and should also document the command-specific options and arguments. Users should be able to access this using the command-suite's `help` command, giving the command name as an argument, like so:

```
$ bin/todo help new
NAME
    new - Create a new task in the task list

SYNOPSIS
    todo [global options] new [command options] task_name

DESCRIPTION
    A task has a name and a priority. By default, new tasks have
    the lowest possible priority, though this can be overridden.

COMMAND OPTIONS
    -f          - put the new task first in the list
    -p priority - set the priority of the new task, 1 being
                  the highest (default: none)
```

This may sound complex; however, open source libraries like GLI actually make this quite simple. Apps that use GLI, like `todo`, include a `help` command

by default, which provides the two-level help system we just described. We can see this in action by running our todo app right now:

```
$ bin/todo help
NAME
    todo -

SYNOPSIS
    todo [global options] command [command options] [arguments...]

GLOBAL OPTIONS
    -f, --filename=arg -
    --help             - Show this message

COMMANDS
    done -
    help - Shows a list of commands or help for one command
    list -
    new  -
$ bin/todo help new
new
```

Like OptionParser, GLI provides the scaffolding and support for the help system and even formats everything for us; we just need to provide the help text for the global options, the commands, their options, and their arguments. This is done in GLI via three methods:

desc	Provides a short, one-line summary of a command or option
long_desc	Provides a more detailed explanation of a command or option (later, we'll talk about the difference between this and the shorter summary you'd put in desc)
arg_name	Gives the argument to a command or flag a short, descriptive name

Once we fill in our app using these methods, our help system will look just like the one shown at the start of this section. Here's what the new command's implementation looks like when fully documented using these methods:

be_helpful/todo/bin/todo

```
➤ desc 'Path to the todo file'
  flag [:f,:filename]
➤ desc 'Create a new task in the task list'
➤ long_desc "
➤ A task has a name and a priority. By default, new
➤ tasks have the lowest possible priority, though
➤ this can be overridden.
➤ "
➤ arg_name 'task_name'
  command :new do |c|
```

```
➤    c.desc 'set the priority of the new task, 1 being the highest'
➤    c.arg_name 'priority'
     c.flag :p

➤    c.desc 'put the new task first in the list'
     c.switch :f
     c.action do |global_options,options,args|
     end
end
```

As you can see, we call desc, long_desc, and arg_name *before* the element they document. This is exactly how Rake works (and also how we document our code; documentation comments appear before the code they document). This keeps our app's code very readable and maintainable.

Now that we've filled this in, our app comes alive with an easy-to-use help system:

```
$ bin/todo help
NAME
    todo -

SYNOPSIS
    todo [global options] command [command options] [arguments...]

GLOBAL OPTIONS
    -f, --filename=todo_file - Path to the todo file (default:
                               /Users/davec/.todo.txt)
    --help                   - Show this message

COMMANDS
    done - Complete a task
    help - Shows a list of commands or help for one command
    list - List tasks
    new  - Create a new task in the task list
$ bin/todo help new
NAME
    new - Create a new task in the task list

SYNOPSIS
    todo [global options] new [command options] task_name

DESCRIPTION
    A task has a name and a priority. By default, new tasks have
    the lowest possible priority, though this can be overridden.

COMMAND OPTIONS
    -f          - put the new task first in the list
    -p priority - set the priority of the new task, 1 being
                  the highest (default: none)
```

One last thing that's worth pointing out is the documentation for the global flag, -f. You'll note that our documentation string includes (default: /Users/davec/.todo.txt). We didn't include that in the string given to desc; it's an additional bit of documentation the GLI derives for us when we use the default_value method to indicate the default value for a flag.

be_helpful/todo/bin/todo
```
desc "Path to the todo file"
arg_name "todo_file"
➤ default_value "#{ENV['HOME']}/.todo.txt"
flag [:f,:filename]
```

default_value isn't actually for documentation; it allows us to specify the value for a flag when the user omits it from the command line; this means that the value of global_options[:f] will not be nil; it will be ~/.todo.txt if the user omits -f on the command line. GLI helpfully includes this in our help text, meaning our documentation and our code will always be consistent.

We've now learned how easy it is to provide help documentation for simple command-line apps and command suites. By adding a few extra strings to our code, our apps can easily help users understand what the apps do and how to use them. But not all apps are so simple. Command-line apps often provide sophisticated behavior that can't be easily explained in the one or two lines of text available in the built-in help systems. How can we provide detailed documentation beyond simple help text?

3.3 Including a Man Page

As we've seen, it's easy to document the options, arguments, and commands of a command-line app. This information, and the ability to access it from the app itself, is invaluable to repeat users of your app; they can quickly find out how to use your app the way they need to get their work done. What if we need more? Perhaps we'd like some longer examples for new users, or perhaps our app is sufficiently complex that we need more space to explain things.

Even a straightforward app like db_backup.rb can benefit from a few examples and some detailed documentation (such as an explanation of the "iteration backup" concept or why the username must be in first.last format). There isn't enough space in the built-in help provided by OptionParser for this information. Furthermore, these are not details that a regular user will need. Frequent users will just want the usage statement and options reference via –help and won't need tutorials, examples, or detailed documentation when they just need to get a list of options.

A traditional UNIX app provides this detailed information in a *manual*, or *man*, page, which users access via the man command. If you type man ls on the command line, you'll see a nice, detailed explanation of the ls command. However, although you could bundle a man page with your Ruby command-line app, man wouldn't be able to access it easily because of the way RubyGems installs apps (we'll talk more about RubyGems in Chapter 7, *Distribute Painlessly*, on page 101). Even if man *could* access your app's files, creating a man page is no small feat; it requires using the nroff[2] format, which is cumbersome to use for writing documentation.

Fortunately, the Ruby ecosystem of open source libraries has us covered. gem-man,[3] a plug-in to RubyGems created by GitHub's Chris Wanstrath, allows users to access man pages bundled inside a gem via the gem man command. ronn[4] is a Ruby app that allows us to create man pages in plain text, without having to learn nroff. We can use these two tools together to create a manual page that we can easily distribute with our app and that will be easily accessible to our users.

Once we've installed these tools, created our man page, and distributed our app to users, they'll be able to read whatever detailed documentation we've provided like so:

```
$ gem man db_backup
DB_BACKUP.RB(1)                                  DB_BACKUP.RB(1)

NAME
       db_backup.rb - backup one or more MySQL databases

SYNOPSIS
       db_backup.rb database_name
       db_backup.rb -u username -p password database_name
       db_backup.rb -i|--iteration database_name
etc....
```

Installing Man Page Tools

Installing gem-man and ronn is straightforward using RubyGems' gem command:

```
$ gem install gem-man ronn
Successfully installed gem-man-0.2.0
Building native extensions.  This could take a while...
Building native extensions.  This could take a while...
Successfully installed hpricot-0.8.4
```

2. http://en.wikipedia.org/wiki/Nroff
3. http://defunkt.io/gem-man/
4. http://rtomayko.github.com/ronn/

```
Successfully installed rdiscount-1.6.8
Successfully installed mustache-0.99.4
Successfully installed ronn-0.7.3
5 gems installed
```

The extra gems installed are gems needed by ronn (we'll talk about runtime dependencies later in Chapter 7, *Distribute Painlessly*, on page 101).

Now that we have our libraries and tools installed, we need to set up a location for our man page's source to live in our project. By convention, this location is a directory called man, and our source file is named APP_NAME.1.ronn (where APP_NAME is the name of our app).

```
$ mkdir man
$ touch man/db_backup.1.ronn
```

Although the directory man is just a convention, the .1 in our filename is required. This number represents the "section" of the manual where our man page will live. The UNIX manual has several sections, and section 1 is for command-line executables.[5] The other part of the name (db_backup) is the name users will use to read our app's manual page. Technically we could call it something else, like foobar, but then our users would need to run gem man foobar instead of gem man db_backup. So, we use the name of our app as the base of the filename.

Now that we have all the pieces in place, let's create our man page.

Creating a Man Page with ronn

We said earlier that ronn allows us to write a man page in plain text, without having to use nroff. This is only partially true. What ronn really does is allow us to use the plain-text format Markdown[6] to write our man page.

Markdown text looks like plain text but actually follows some lightweight conventions for formatting lists, calling out sections, and creating hyperlinks. It's much simpler than HTML and a lot easier to create than nroff. The ronn-format documentation[7] provides a comprehensive reference for the Markdown syntax relevant to a man page. Text formatted in Markdown is actually quite simple, so let's take a look at some.

Here's what a man page for db_backup.rb looks like:

5. The Wikipedia entry for the UNIX man system (http://en.wikipedia.org/wiki/Man_page#Manual_sections) has a good overview of the other sections if you are interested.
6. http://daringfireball.net/projects/markdown/
7. http://rtomayko.github.com/ronn/ronn-format.7.html

be_helpful/db_backup/man/db_backup.1.ronn

```
db_backup.rb(1) -- backup one or more MySQL databases
=====================================================

## SYNOPSIS

`db_backup.rb` <database_name><br>
`db_backup.rb` `-u username` `-p password` <database_name><br>
`db_backup.rb` `-i`|`--iteration` <database_name>

## DESCRIPTION
**db_backup.rb** is a simple command-line tool for backing up a
MySQL database.  It does so safely and quietly, using a sensible
name for the backup files, so it's perfect for use with cron as
a daily backup.

By default, `db_backup.rb` makes a daily backup and names the
resulting backup file with the date.  `db_backup.rb` also
understands our development process, so if you specify the
`--iteration` flag, the backup will be named differently than
for a daily backup.  This will allow you to easily keep one
backup per iteration, easily identifying it, and differentiate
it from daily backups.

By default, `db_backup.rb` will use your database credentials
in `~/.my.cnf`, however, you can override either the username
or password (or both) via the `-u` and `-p` flags, respectively.
Finally, `db_backup.rb` will add a sanity check on your username, to
make sure it fits with our corporate standard format of `first.last`.

## FILES

`~/.my.cnf` is used for authentication if `-u` or `-p` is omitted.

## OPTIONS

  * `-i`, `--iteration`:
    Indicate that this backup is an "end of iteration" backup.
  * `-u USER`:
    Database username, in first.last format
    `~/my.cnf` is not correct
  * `-p PASSWORD`:
    Database password

## EXAMPLES

Backup the database "big_client"

    $ db_backup.rb big_client
```

```
Backup the database "small_client", for which different credentials are required:

    $ db_backup.rb -u dave -p d4v3 small_client

Make an iteration backup of the "big_client" database:

    $ db_backup.rb -i big_client
```

The formatting reads very well just as plain text, but the Markdown format tells ronn things like this:

- ## marks the beginning of a new section.
- A string like **db_backup.rb** should be displayed in bold.
- Paragraphs preceded by asterisks are a bullet list.

Content-wise, we've replicated some of the information from our code to OptionParser, and we've expanded on a few topics so that a newcomer has a lot more information about how things work. We've also taken advantage of the standard sections that might appear in a man page so that experienced users can quickly jump to the section they are interested in. We'll talk about what sections you might want to include on a man page in the final part of this chapter.

To actually generate our man page from the Markdown source, we use ronn as follows:

```
$ ronn man/db_backup.1.ronn
     roff: man/db_backup.1
     html: man/db_backup.1.html
```

ronn also generates an HTML version suitable for including on your app's website. To preview our man page as command-line users will see it, we can use the UNIX man command on the nroff file generated by ronn:

```
$ man man/db_backup.1
DB_BACKUP.RB(1)                                DB_BACKUP.RB(1)

NAME
       db_backup.rb - backup one or more MySQL databases

SYNOPSIS
       db_backup.rb database_name
       db_backup.rb -u username -p password database_name
       db_backup.rb -i|--iteration database_name
```

We've omitted most of the man page content for brevity, but you can see that it's nicely formatted like any other UNIX man page. To have this man page distributed with our app, we'll need to learn more about RubyGems, which

we'll do later in Chapter 7, *Distribute Painlessly*, on page 101. For now, we'll just tell you that if you include this file in your gem and another user installs your app via RubyGems, users will be able to read your man page right from the command line.[8]

```
$ gem man db_backup
DB_BACKUP.RB(1)                                    DB_BACKUP.RB(1)

NAME
       db_backup.rb - backup one or more MySQL databases

SYNOPSIS
       db_backup.rb database_name
       db_backup.rb -u username -p password database_name
       db_backup.rb -i|--iteration database_name
```

This, combined with the great built-in help that OptionParser or GLI gives you, will ensure that your app is helpful to all users, allowing them to easily and quickly understand how to use your app. You'll be free to focus on what your app does instead of formatting and maintaining documentation.

We now know the nuts and bolts of creating help and documentation, but it's worth having a brief discussion on style. There remain a few aspects of help that are "fuzzy" but nevertheless important, and knowledge of a few more documentation conventions will help you write great documentation without being too verbose.

3.4 Writing Good Help Text and Documentation

That your app has any help text at all is great and puts it, sadly, ahead of many apps in terms of ease of use and user friendliness. We don't want to be merely great; we want to be awesome, so it's important that our help text and documentation be clear, concise, accurate, and useful. We're not going to get into the theory of written communication, but there are a few rules of thumb, as well as some formatting conventions, that will help elevate our help text and documentation.

In general, your in-app help documentation should serve as a concise reference. The only portion of the in-app help that needs to be instructive to a newcomer is the "banner," that is, the one-sentence description of your program. Everything else should be geared toward allowing regular users of your program to remember what options there are and what they do.

8. Savvy users can alias man to be gem man -s, which tells gem-man to use the system manual for any command it doesn't know, thus providing one unified interface to the system manual and the manual of installed Ruby command-line apps.

Anything else should go into your man page and should include information useful to a newcomers (particularly the "DESCRIPTION" and "EXAMPLE" sections), examples, and more in-depth information for advanced users who want to dig deeper.

Let's walk through each element of our app's documentation and discuss how best to write it.

Documenting an App's Description and Invocation Syntax

The first thing a user will expect to see is the banner, which, in the case of db_backup.rb, contains a one-line description of the app, along with its invocation syntax. This description should be one *very* short sentence that sums up what the app does. If it's longer than sixty characters, it's probably too long, and you should try to summarize it better. (The number sixty is based on a standard terminal width of eighty characters; the difference of twenty characters gives you plenty of breathing room for the name of the app and some whitespace, but in general it forces you to be concise, which is a good thing.)

The invocation syntax or "usage" should follow a fairly strict format. For non-command-suite apps, it will be as follows:

```
«executable» [options] «arg_name1» «arg_name2»
```

where *«executable»* is the name of your executable and *«arg_name1»* and *«arg_name2»* are the names of your arguments. Note that [options] should be included only if your app takes options; omit it if it doesn't.

For a command suite, the format is very similar; however, you need to account for the placement of the command and the differentiation of global and command-specific options. GLI's behavior here is what you want:

```
«executable» [global options] «command» [command options] «arg_name1» «arg_name2»
```

Here, *«command»* is the command being executed. Much like a simple command-line app, you should omit [global options] if your command suite doesn't take global options and omit [command options] if the particular command doesn't take command-specific options.

In other cases, the arguments' names should be brief, with multiple words separated by underscores. If your app requires a lot of arguments, this may be an indicator that you have not designed your UI humanely; you should consider turning those arguments into flags.

If your app takes multiple arguments of the same type (such as a list of files on which to operate), use an ellipsis, like so: arg_name...

The ellipsis is a common indicator that one or more arguments of the same type may be passed, so, in the case of our database backup app, since we accept multiple database names, we should use database_name... to indicate this.

You also might be wondering why the options are documented using square brackets. This is because options are optional, and the UNIX convention is to show optional elements in square brackets. If you recall from Table 1, *Overview of OptionParser parameters to on*, on page 21, a string like "-n [NAME]" indicates to OptionParser that the argument NAME to the flag -n is optional. The reason OptionParser uses square brackets is because of this convention.

It might be possible that your app requires certain options to always be set on the command line; this is discouraged and will be discussed in more detail in Chapter 5, *Delight Casual Users*, on page 71.

You can use the square-bracket syntax for a command's arguments as well. You should use [file_name] to indicate one file can be specified but that it isn't required, or you can use [file_name...] to indicate zero or more files would be accepted.

Documenting Options

Like the summary description of an app, one brief summary sentence should be sufficient to document each of its flags and switches. Each sentence should be as clear and to the point as possible. Again, sixty characters is a good limit to set, though it might be harder to hit for more complex options.

An argument to a flag should have a short, one-word description (or, if using multiple words, each separated with an underscore). If a flag has a default value, provide that value in the help string as well (we saw this in the documentation of the global flag -f that todo uses to locate the task list file). As with arguments for your app, if the argument to your flag is optional, surround its name with square brackets.

OptionParser and GLI more or less handle the formatting part for you, so you mainly need to focus on keeping your descriptions brief and to the point.

Documenting Commands in a Command Suite

Commands for a command suite require two bits of documentation: a one-line summary for the first level of help (e.g., the help shown by a command such as your_app help) and a longer description for the second level of help (e.g., from your_app help command_name).

As you might be able to guess by now, the one-line summary should be a very short description of what that command does; you can elaborate in the longer description. The GLI desc and long_desc methods provide a place for this documentation.

Command-specific arguments should follow the same conventions as those we discussed for the app's usage statement. The arg_name method in GLI provides a place to do this per-command.

Documenting Everything Else

An app's man page (or other extra documentation) provides more detail about how the app works and how its various options and arguments affect its behavior. This document should be sorted into sections, which will help to keep it organized and navigable (experienced UNIX users will look for certain section names so they can quickly scan the documentation). Table 2, *Common sections for your gem-man pages* outlines the common sections you might need and what goes in them (though you aren't limited to these sections; use your best judgment). Note that they are also listed *in the order that they should appear* in the man page and that you should include, at the very least, a "SYNOPSIS," "DESCRIPTION," and "EXAMPLE."

Section	Meaning
SYNOPSIS	A brief synopsis of how to invoke your app on the command line. This should be similar to what the default banner is in OptionParser or what is output by GLI after the "Usage:" bit.
DESCRIPTION	A longer description of what your app does, why the user might use it, and any additional details. This section should be targeted at new users and written to help them understand how to use your app.
OPTIONS	A bullet list documenting each option. This is a chance to explain in more detail how the options work and what effect they have on your app's behavior.
EXAMPLES	One or more examples of using the app, including brief text explaining each example.
FILES	A list of files on the filesystem that the app uses (for example, todo would document the default to-do list file's location here).

Section	Meaning
ENVIRONMENT	A bullet list of any environment variables that affect the app's behavior.
BUGS	Known bugs or odd behaviors.
LICENSE	The name of the license and a reference to the full license text (you do not need to reproduce the entire license here).
AUTHOR	The authors of your app and their email addresses.
COPYRIGHT	The copyright information for your app.
SEE ALSO	Links to other commands or places on the Web that are relevant for your app.

Table 2—Common sections for your gem-man pages

You should reuse documentation from your command-line interface here; typically your one-line summaries will be the first sentence of the paragraph that documents something. This saves you time, and it also helps connect the built-in help text to the more detailed documentation. Unfortunately, neither OptionParser nor GLI provide a way to autogenerate a man page so that your documentation can automatically be kept consistent. Perhaps you'll be inspired and create such a system.

3.5 Moving On

You should now know everything you need to know to make a command-line application that's helpful to newcomers as well as experts. All this is good news for your users; they'll have an easy time using your apps, but it's also good news for you as the developer; you can spend more time on your apps' actual functionality and less on formatting help text and documentation.

Users aren't the only entities who interact with your application, however. The system itself will be actually executing your application, and future developers may need to integrate your command-line apps into larger systems of automation (similar to how our database backup script integrates mysqldump). In the next chapter, we'll talk about how to make your apps interoperate with the system and with other applications.