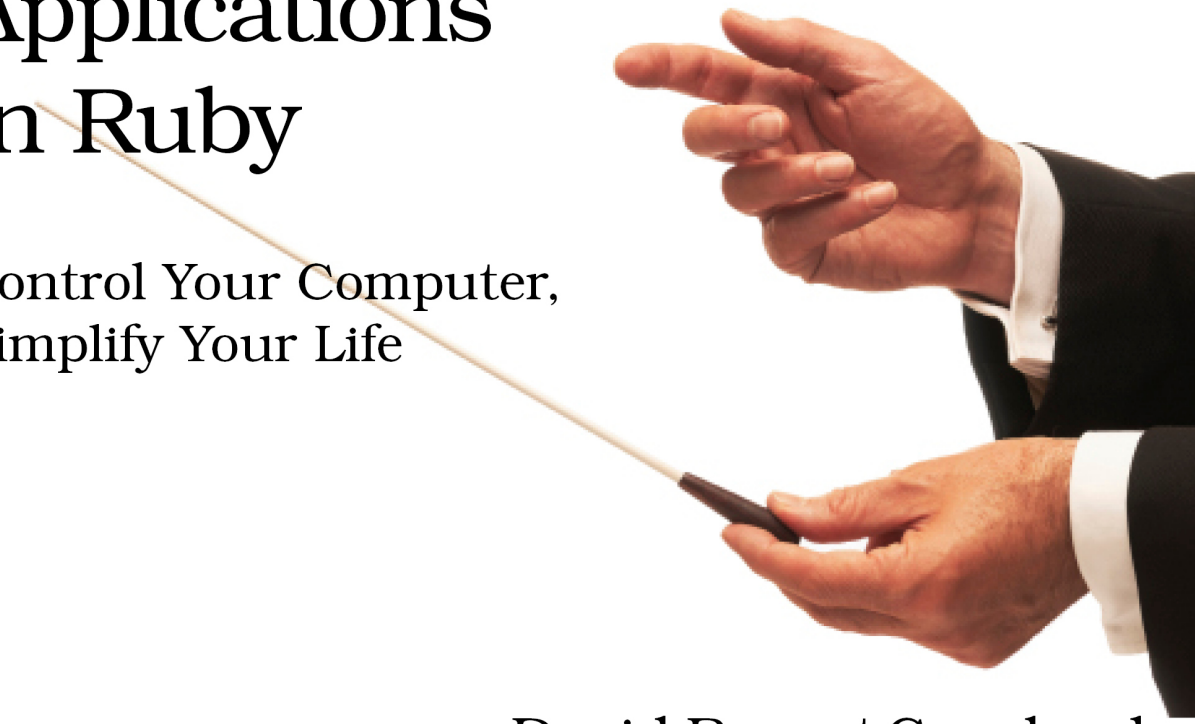
The Pragmatic Programmers

# Build Awesome Command-Line Applications in Ruby

## Control Your Computer, Simplify Your Life



David Bryant Copeland

*Edited by John Osborn*

## What Readers Are Saying About
## *Build Awesome Command-Line Applications in Ruby*

Some command-line applications save time and are a joy to use. Others just make you want to tear your hair out. David Copeland has written a guide to writing the kind of command-line apps that will make your users want to hug you. From providing a humane command-line interface, to being self-documenting, to integrating seamlessly with the rest of the command-line universe—this book will show you how to take your scripts from adequate to awesome.

➤ **Avdi Grimm**
Ruby developer, author, *Exceptional Ruby*, and blogger, Virtuous Code

This book proves that text mode is not the just the domain of batch scripts and glue code. Beyond the extensive survey of current Ruby CLI tools, David brings an unmatched focus on user experience and testing. Every full-stack developer should learn how to build the kinds of apps covered in this book.

➤ **Wynn Netherland**
CTO, Pure Charity

I know of no other Ruby book that covers the content in this useful work, especially with its eye toward making Ruby command-line applications better citizens.

➤ **Noel Rappin**
Senior engineer at Groupon and author, *Rails Test Prescriptions*

This well-written book teaches ideas that are really important: that Ruby is a powerful language for writing command-line tools; that CLI tools, unlike GUI tools, can be combined in an infinite number of ways; that the effort required to automate small recurrent tasks pays off; and that there are time-tested best practices for succeeding with command-line tool development. Not only are the scripts in this volume awesome, so is the book.

➤ **Staffan Nöteberg**
Author, *Pomodoro Technique Illustrated*

I want a few people on my team to have this book now. I especially can't wait to get this in the hands of our software lead, who's a whiz at shell scripts and would be delighted to see how much easier and more reliable option parsing is in Ruby.

➤ **Ian Dees**
Ruby developer and coauthor, *Using JRuby*

This book teaches you how to write command-line tools your mother would be proud of.

➤ **Matt Wynne**
Independent consultant, programmer, coach, and author, *The Cucumber Book*

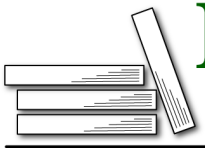
# Build Awesome Command-Line Applications in Ruby

## Control Your Computer, Simplify Your Life

David Bryant Copeland

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

John Osborn (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)



Printed in the United States of America.
ISBN-13: 978-1-934356-91-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

# Contents

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 15. 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.

## 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
```

*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

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

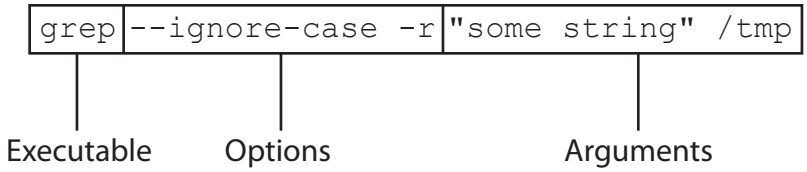



**Figure 1—Basic parts of a command-line app invocation**

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 16. 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.

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 15, arguments are the elements of a command line that aren't options. Rather,

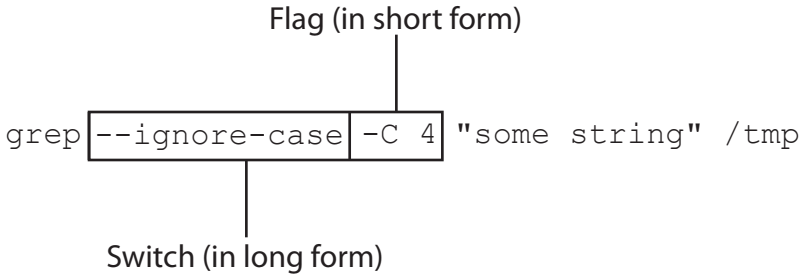



**Figure 2—A command-line invocation with switches and flags**

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 15 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 18 shows a command-suite invocation, with the command's position on the command line highlighted.

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 18.

*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.

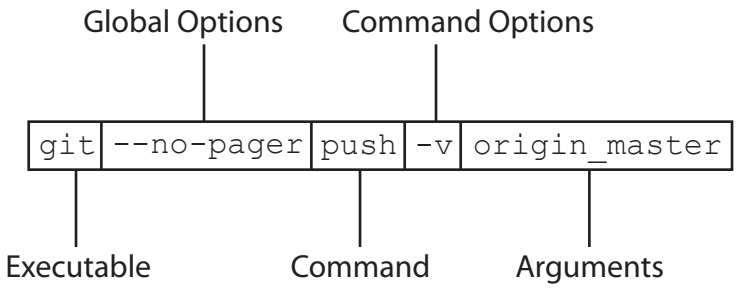



**Figure 3—Basic parts of a command-suite invocation**

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.

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

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? 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:/www.ruby-doc.org/stdlib/libdoc/optparse/rdoc/files/optparse_rb.html#M001903

| 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

## Type Conversions in OptionParser

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:

```
ops.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, DecimalInteger, 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,blah" yields the list ["foo","bar","blah"].

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) 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:quux will result in custom_attributes getting the value { 'foo' => 'bar', 'baz' => 'quux' }.

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).

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

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.

### 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-1.3.0
1 gem installed
$ gli scaffold to-do 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/tc_nothing.rb
Created ./todo/Rakefile
Created ./todo/Gemfile
Created ./todo/lib/todo_version.rb
```

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:

```
$ bin/todo new
$ bin/todo done
$ bin/todo list
$ 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. Let's now look at the code GLI produces to see how it works. Note that GLI has generated only the code it needs to parse the commands we passed as arguments to the scaffold command. The switches and flags GLI sets up are merely examples; we'll see 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 'rubygems'
require 'gli'
include GLI
```

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
```

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.com/gli/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 GLI.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 "-p - #{options[:p]}"
      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:

```
$ 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 8

# Test, Test, Test

Writing perfect code isn't easy. It might even be impossible. That's why seasoned developers—including command-line application developers—write tests. Tests are the best tool we have to make sure our applications perform flawlessly, and the Ruby community is especially friendly to testing, thanks to the culture and tools established by Ruby on Rails.

Command-line applications often interact with various systems and environments, which produces a unique set of challenges for testing. If you've done any web application development, you are probably accustomed to having different "tiers," such as a development tier and a testing tier. These tiers are complete systems, often using many servers to provide an environment for isolated testing. This is impractical for command-line apps, so we tend to develop command-line apps on the system where they are intended to run. Therefore, if we wanted to test adding a task to our task list using `todo`, it would add a task to our actual task list, if we didn't take steps to keep it from doing so.

What we'll learn here is how to write and run tests for our command-line apps, as well as some techniques to keep our tests from causing problems on our system. We'll do this by combining two types of tests: unit tests and acceptance tests. You are probably familiar with unit testing, which is useful in testing the small bits of logic that comprise our application. Ruby's standard library includes all the tools we'll need.

Acceptance testing takes the opposite approach. Acceptance tests simulate real user behavior and exercise the entire system. We'll learn about this type of testing by using the popular Cucumber testing tool, along with Aruba, which is an extension to Cucumber designed to help test command-line applications. Acceptance tests are a good place to start because of the user-centered approach, so let's jump in and see how they work. After we have a

good set of acceptance tests, we'll turn our attention to unit tests to test edge cases that might be hard to simulate using acceptance tests.

## 8.1 Testing User Behavior with Acceptance Tests

Unlike web applications, command-line apps have simple user interfaces, and their output is less complex. This makes the job of simulating input and testing output relatively simple. The problems come in when we consider what our apps do. `db_backup.rb` takes a long time to do its work and requires access to a database. `todo` makes irreversible changes to its task list. We need a way to run our apps that mimics as closely as possible the "real-world" scenarios they were written to solve but in a repeatable and predictable way that doesn't cause permanent changes to our environment.

Rather than solve this problem at the same we learn about the mechanics of testing, let's take things one step at a time. If you recall, `todo` takes a global option that controls where the task list is. We can use that to keep our tests from messing with our personal task list in our home directory. Let's use that to get some tests going. After we see how to test our app in general, we'll discuss some techniques to deal with the "tests messing with our personal task list" issue.

### Understanding Acceptance Tests

*Acceptance tests* are tests that we can use to confirm that an app properly implements certain features, from a user's perspective. Acceptance tests typically test only the subset of the actions users are likely to attempt (the so-called happy path), and they don't cover uncommon edge cases. What acceptance tests should do is to simulate the ways users are most likely to employ our tool to do their job. A `todo` user, for example, is likely to do the following:

- Add a new task
- List tasks
- Complete a task
- Get help

Each task maps to a command that `todo` accepts. To test these, we need to use `todo` just as a user would. We can also use `todo` to verify its own behavior. For example, we can execute a `todo list`, capture the list of tasks, add a new task via `todo new`, and then list the tasks again, this time looking for our new task. We could use the same technique to test `todo done`.

While we could create *another* command-line app to run these tests, we don't need to do so. The acceptance testing tool Cucumber (explained in great detail in *The RSpec Book* [CADH09]) can handle the basic infrastructure for running tests, and the Cucumber add-on library Aruba will provide us with the tools we need to run our app and verify its behavior. First we'll see how these tools work, then we'll set them up, and finally we'll write our tests.

### Understanding Cucumber and Aruba

Acceptance tests written in Cucumber don't look like the tests you might be used to seeing. They're written in what looks like plain English.[1] With Cucumber, you describe the behavior you want to test, in English, and then write code that runs under the covers to execute the procedure you have specified.

Now, you can't just write free-form text; there *is* a structure that you'll need to follow to make this work. Cucumber delineates tests into *features*, which contain multiple *scenarios*. A feature is what it sounds like: a feature of your application. "Adding a task" is a feature. A scenario exercises an aspect of a feature. For example, we might have a scenario to add a task to an existing task list and another to add a task for the very first time.

In Cucumber, you describe the feature in free-form English; this part of the test is mere documentation. The scenarios, however, must follow a strict format. Each scenario has a one-line description and is followed by *steps*. Steps start with "Given," "When," "Then," "And," or "But," for example "Given the file /tmp/todo.txt exists" or "Then the output should contain Hello." These steps are what Cucumber will actually execute to run your test. Before we get into the weeds of how this works under the covers, let's look at a simple feature and scenario for `todo`:

tolerate_gracefully/todo/features/todo.feature

```
Feature: We can add new tasks
  As a busy developer with a lot of things to do
  I want to keep a list of tasks I need to work on

Scenario: Add a new task
  Given the file "/tmp/todo.txt" doesn't exist
  When I successfully run `todo -f /tmp/todo.txt new 'Some new task'`
  Then I successfully run `todo -f /tmp/todo.txt list`
  And the stdout should contain "Some new task"
```

1. Cucumber supports other human languages as well, from Arabic to Vietnamese.

As you can see, we've followed Cucumber's particular format, but the test is written in plain English. You could take these instructions and manually execute them on the command line to check that adding a task works correctly.

Seeing the text of this test gives us some insight as to the use of "Given," "When," and "Then." These three words help to differentiate the three main parts of any good test: setup, action, and verification. More precisely:

Given
: This sets up the conditions of the test. Most tests operate under a set of assumptions, and each "Given" step establishes what those are. In our case, we are establishing that the task list doesn't exist so that when we later add a task to it, we can be sure that our action had an effect (if we didn't have this bit of setup, a tasklist might exist that contains the task we're adding, resulting in our test always passing, even if our app was broken).

When
: This performs the action or actions under test. In our case, we run `todo new` to, ideally, add a new task.

Then
: This verifies that our action taken in a "When" had the desired outcome. For our earlier scenario, we verify that `todo new` worked by running `todo list` and examining its output.

And or But
: These two words can be used anywhere and "extend" the keyword they follow. We're using that in our "Then" section, because we need two steps to perform a full verification. The use of "And" over "But" is purely cosmetic; use whatever "reads" the best to you.

Back to our scenario, we now have a set of steps that, as we mentioned, we could manually execute to test our app. Of course, we don't want to have to run our tests manually; we're using Cucumber to automate all of this, so let's dig a bit deeper to see how Cucumber can run this test. We'll get into the specifics of where files go and what their names are, so for now, let's assume that we can ask Cucumber to run this feature for us.

The first step of our scenario is `Given the file "/tmp/todo.txt" doesn't exist`. This is a setup step that ensures we don't have a tasklist sitting around from a previous test run that might already have our about-to-be-added task in it. Cucumber doesn't know how to perform this setup step, so we need to give it the code

to do so. We'll define the step using the method Given, provided by Cucumber, that takes a regexp and a block. If the text of our step matches that regexp, the block is executed. Let's see this step's definition:

tolerate_gracefully/todo/features/step_definitions/cli_steps.rb
```
Given /^the file "([^"]*)" doesn't exist$/ do |file|
  FileUtils.rm(file) if File.exists? file
end
```

You'll notice that the regexp has a capture in it (this part: ([^"]*)). Cucumber will extract whatever matched that capture and provide it as an argument to our block. This means we could write another step, Given the file "/tmp/some_other_file.txt" doesn't exist, and this step definition would work for it, too. Using this technique, you could build up a library of reusable step definitions. This is exactly what Aruba is: a set of general-purpose step definitions for writing scenarios to test command-line apps.

The next two steps of our scenario, both of the form "I successfully run 'some command,'" are defined by Aruba, meaning we don't have to provide step definitions for those steps. This also demonstrates an interesting aspect of Cucumber. At a code level, Given, When, Then, And, and But are all treated the same. We could have every step start with "And" and Cucumber wouldn't care; these keywords are for human eyes, not the computer. Further, when we define steps, Cucumber provides Given, When, and Then to do so, but the effect is the same regardless of which one we use. This is how we're able to use the same step as both a "When" and a "Then."

Coming back to the Aruba-provided step "I successfully run 'some command,'" this does two things: it executes the command in the backticks, and it checks its exit status. If it's zero, the test will proceed. If it's nonzero, the test will halt with a failure (there is a less stringent version provided, "I run 'some_command,'" that will just run the command and not check the exit status). The final step is also defined by Aruba and allows us to assert that the standard output of our app's run contains a particular string.

Now that you have an idea of what Cucumber is and a general sense of how it works, let's actually get it set up for our project so we can run this feature.

### Installing and Setting Up Cucumber and Aruba

To install Cucumber and Aruba, we need only add the aruba gem to our gemspec (as a development dependency; our app doesn't require aruba to run). Because Aruba depends on Cucumber, when you update your gems with Bundler, Cucumber will be installed automatically (see the note on Windows at *Using Aruba on Windows*, on page 122). Here's the updated gemspec:

### Using Aruba on Windows

As of this writing, Aruba does not work "out of the box" on Windows. Discussions on the Internet involve varying degrees of functionality, but there doesn't seem to be a strong consensus that Aruba works on Windows.

If you are adventurous, I encourage you to attempt to get the Cucumber tests in this section passing on Windows and submit a patch. Windows is an important operating system for the Ruby community that is, currently, sorely underrepresented in compatibility. This particular application, spawning processes and monitoring their behavior, is particularly tricky, because of the large differences between UNIX and Windows. Since OS X is based on UNIX, Macs tend to work just fine.

I still encourage you to follow along with this section, because it will teach you solid principles on acceptance testing your apps; you may need to take a different approach until Aruba is more compatible with Windows.

tolerate_gracefully/todo/todo.gemspec

```
spec = Gem::Specification.new do |s|
  s.name = 'todo'
  s.version = Todo::VERSION
  # rest of the gemspec...

  s.bindir = 'bin'
  s.executables << 'todo'
➤ s.add_development_dependency('aruba', '~> 0.4.6')
  s.add_dependency('gli')
end
```

Now, we tell Bundler to make sure our development dependencies are up-to-date:

```
$ bundle install
Using ffi (1.0.9)
Using childprocess (0.1.9)
Using builder (3.0.0)
Using diff-lcs (1.1.2)
Using json (1.5.1)
Using gherkin (2.4.0)
Using term-ansicolor (1.0.5)
Using cucumber (0.10.6)
Using rspec-core (2.6.4)
Using rspec-expectations (2.6.0)
Using rspec-mocks (2.6.0)
Using rspec (2.6.0)
Using aruba (0.3.7)
Using gli (1.3.2)
Using todo (0.0.1) from source at .
Using bundler (1.0.17)
```

Cucumber has a conventional file structure where the directory features (off of the project's root directory) contains the .feature files containing our scenarios. Inside that directory, the directory step_definitions contains the Ruby code to define our steps. The names of the files don't matter; everything with an .rb extension will be loaded. Finally, any Cucumber configuration goes in support/env.rb. We'll see what this file is for later.

Finally, we need to add a task to our Rakefile to allow us to run the Cucumber scenarios:

tolerate_gracefully/todo/Rakefile

```
require 'cucumber'
require 'cucumber/rake/task'

Cucumber::Rake::Task.new(:features) do |t|
  t.cucumber_opts = "features --format pretty -x"
  t.fork = false
end
```

Now that we've seen what our tests look like, implemented the needed steps, and installed all the software we need, let's run our tests.

### Running Cucumber Tests

The task we added to our Rakefile creates a task named "features" that will run our Cucumber tests. Let's run it now:

```
$ rake features
Feature: We can add new tasks
  As a busy developer with a lot of things to do
  I want to keep a list of tasks I need to work on

  Scenario: Add a new task
    Given the file "/tmp/todo.txt" doesn't exist
    When I successfully run
      `todo --filename=/tmp/todo.txt new 'Some new todo item'`
    And I successfully run `todo --filename=/tmp/todo.txt list`
    Then the stdout should contain "Some new todo item"
1 scenarios (1 passed)
4 steps (4 passed)
0m0.563s
```

If you're running this locally, you'll notice that the output is green. This means that everything is working and our test passed. Let's introduce a bug to see what happens when our tests fail. Here's our original, correct, CSV formatting code that we saw in Chapter 4, *Play Well with Others*, on page 53:

play_well/todo/bin/todo
```
complete_flag = completed ? "C" : "U"
printf("%d,%s,%s,%s,%s\n",index,name,complete_flag,created,completed)
```

We'll uppercase the name of the task in our CSV output, which should cause our test to fail, like so:

```
elsif options[:format] == 'csv'
  # Use the machine-readable CSV format
  complete_flag = completed ? "C" : "U"
➤ printf("%d,%s,%s,%s,%s\n",index,name.upcase,complete_flag,created,completed)
end
```

Now, when we run our feature, one of the steps in our scenario fails:

```
$ rake features
Scenario: Add a new task
  Given the file "/tmp/todo.txt" doesn't exist
  When I successfully run
    `todo --filename=/tmp/todo.txt new 'Some new todo item'`
  And I successfully run `todo --filename=/tmp/todo.txt list`
  Then the stdout should contain "Some new todo item"
  expected "1,SOME NEW TODO ITEM,U,Thu Sep 22 08:44:02 -0400 2011,\n"
    to include "Some new todo item"
  Diff:
  @@ -1,2 +1,2 @@
  -Some new todo item
  +1,SOME NEW TODO ITEM,U,Thu Sep 22 08:44:02 -0400 2011,
   (RSpec::Expectations::ExpectationNotMetError)
  features/todo.feature:17:in `Then the stdout should
       contain "Some new todo item"'

Failing Scenarios:
cucumber features/todo.feature:13 # Scenario: Add a new task

1 scenarios (1 failed)
4 steps (1 failed, 3 passed)
0m0.576s
rake aborted!
Cucumber failed
```

If you're running this locally, all of the error text will be displayed in red, giving a clear indication that something is wrong. If you look closely, you'll notice that Aruba has provided us with a diff of the expected output and received output, making it fairly easy to see the problem that caused our test to fail.

## Testing Complex Behavior

Now that we have a basic path tested, let's see how to test something a bit trickier: the default location of the task list. In our previous scenario, we used the --filename option to explicitly control where todo looked for the task list. This is important, because we don't want our tests to mess with our actual task list, which lives in our home directory. Nevertheless, we *do* need to test that todo correctly uses the task list in our home directory by default. This presents us with a problem.

Testing db_backup.rb presents a similar problem; we need a real database to back up, and backing up a database potentially takes a long time. These are two examples of the challenges we face when testing command-line apps. There's no silver bullet to solve these, but if we think creatively, we can handle most of them. To gain insight into how to approach problems like this in the future, let's write tests for both todo's task list defaulting to our home directory and db_backup.rb backing up a real database.

### Testing Access to the Home Directory

It's great that we have the --filename flag to todo; we can get a lot of test coverage without worrying about files in our actual home directory. We *do* need to verify that the default location for the task list gets used. How can we do this without having our tests modify our actual task list?

First let's write the scenario to test what we want and work from there.

tolerate_gracefully/todo/features/todo.feature

```
Scenario: The task list is in our home directory by default
  Given there is no task list in my home directory
  When I successfully run `todo new 'Some new todo item'`
  Then the task list should exist in my home directory
  When I successfully run `todo list`
  Then the stdout should contain "Some new todo item"
```

This should be a good test of the default location; we omit the --filename options, check that a file exists in our home directory, and then use the list command to make sure we're reading from the right place. Before we see how to keep our home directory safe from our tests, let's define all of our steps.

There are two steps in this scenario that we don't have defined. We have steps similar to them; we've implemented the file "xxx" doesn't exist, and Aruba provides the step a file named "xxx" should exist. We can use these steps when making our own, like so:

tolerate_gracefully/todo/features/step_definitions/cli_steps.rb
```
Given /^there is no task list in my home directory$/ do
  step %(the file "#{ENV['HOME']}/.todo.txt" doesn't exist)
end

Then /^the task list should exist in my home directory$/ do
  step %(a file named "#{ENV['HOME']}/.todo.txt" should exist)
end
```

You'll note that we're using ENV['HOME'], which is how we access the HOME environment variable. The system sets this variable to the user's home directory (even on Windows). Assuming that our app uses this to access the user's home directory, we can change its value to another directory that we control. Our tests and the app are still accessing "the user's home directory" in a canonical way, but we can control the contents of that location.

Since apps that Aruba runs inherit the environment of the tests, all we need to do is modify the value of ENV['HOME'] before our scenario runs (and restore its correct value after the scenario exits). Cucumber provides hooks to do just that. The methods Before and After both accept blocks that will execute before and after (respectively) every scenario.

tolerate_gracefully/todo/features/support/env.rb
```
Before do
  @real_home = ENV['HOME']
  fake_home = File.join('/tmp','fake_home')
  FileUtils.rm_rf fake_home, :secure => true
  ENV['HOME'] = fake_home
end
After do
  ENV['HOME'] = @real_home
end
```

As you can see, we create a fresh, empty directory and point ENV['HOME'] to it. Our app uses that same variable like so:

tolerate_gracefully/todo/bin/todo
```
desc "Path to the todo file"
arg_name "todo_file"
default_value File.join(ENV['HOME'],'.todo.txt')
flag [:f,:filename]
```

So, we're able to verify the logic of the task list defaulting to our home directory, without actually using our home directory. Since the location of the "home directory" is really just shorthand for "whatever directory is in ENV['HOME'],"  we now have test coverage without worrying that our personal task list will be touched. Let's run our new test and make sure it passes. To

prove that we aren't touching our home directory, we'll list our actual task list before and after running our feature.

```
$ todo list
1 - Design database schema
    Created:   Sun Oct 02 08:06:12 -0500 2011
2 - Get access to production logs
    Created:   Sun Oct 02 08:06:12 -0500 2011
3 - Code Review
    Created:   Sun Oct 02 08:06:12 -0500 2011
$ rake features
Feature: We can add new tasks
  As a busy developer with a lot of things to do
  I want to keep a list of tasks I need to work on

  Scenario: The task list is in our home directory by default
    Given there is no task list in my home directory
    When I successfully run `todo new 'Some new todo item'`
    Then the task list should exist in my home directory
    When I successfully run `todo list`
    Then the stdout should contain "Some new todo item"

1 scenarios (1 passed)
5 steps (5 passed)
0m0.793s
$ todo list
1 - Design database schema
    Created:   Sun Oct 02 08:06:12 -0500 2011
2 - Get access to production logs
    Created:   Sun Oct 02 08:06:12 -0500 2011
3 - Code Review
    Created:   Sun Oct 02 08:06:12 -0500 2011
```

Our test passed, but our task list wasn't modified—everything worked!

Manipulating the environment is a great technique for testing behavior like this; the environment works as a "middleman" that allows us to change things (like the location of the user's home directory) without affecting the code of our tests or our app. What about testing that certain external commands were called, as in the case of `db_backup.rb`?

### Testing Execution of External Commands

`db_backup.rb` is basically a specialized wrapper around `mysqldump`. If we ran `db_backup.rb` from a Cucumber test as is, it would require a live database and would perform an actual backup. This could be a problem, especially if we ask it to back up a particularly large database.

We could use environment variables again, by setting a special variable that tells db_backup.rb to not actually call mysqldump but instead just print out the command it would normally run.

```
def run(command,exit_on_error_with)
  puts "Running '#{command}'"
➤ unless ENV['DONT_RUN']
    stdout_str, stderr_str, status = Open3.capture3(command)
    puts stdout_str
    unless status.success?
      STDERR.puts "There was a problem running '#{command}'"
      STDERR.puts stderr_str
      exit exit_on_error_with
    end
  end
end
```

This isn't a very good technique; we want to test that we're calling mysqldump appropriately, and doing something like this skips it entirely. We really should test the entire system from end to end.

An acceptance test of the complete system will give you the best idea of how the app will behave in the hands of users. It's also the most difficult to set up, since it requires a completely controlled testing environment. For the purposes of db_backup.rb, we can set up a database for testing, populate it with a small amount of data, and then run our app, checking that it did what we expected.

First let's write out our scenarios. In this case, we'll run two tests: one for the normal use (where the backup is compressed) and one where we do not compress the backup.

tolerate_gracefully/db_backup/features/system_test.feature

```
Scenario: End-to-end test using a real database
  Given the database backup_test exists
  When I successfully run `db_backup.rb --force -u root backup_test`
  Then the backup file should be gzipped

Scenario: End-to-end test using a real database, skipping gzip
  Given the database backup_test exists
  When I successfully run `db_backup.rb --force -u root --no-gzip backup_test`
  Then the backup file should NOT be gzipped
```

This should look familiar by now. As we've seen, Aruba provides the second step of these scenarios for us, but the rest we have to define. First we'll define our "Given," which sets the stage for our test. We need to set up an entire database in MySQL with *some* data in it. To do that, we'll create a .sql file that will set up our database, create a table, and insert some data:

tolerate_gracefully/db_backup/setup_test.sql
```
drop database if exists backup_test;
create database backup_test;
use backup_test;

create table test_table(
    id int,
    name varchar(255)
);

insert into test_table(id,name) values (1,'Dave'), (2, 'Amy'), (3,'Rudy');
```

We include that in our project and can now reference it in our step definition, which looks like so:

tolerate_gracefully/db_backup/features/step_definitions/system_test_steps.rb
```
MYSQL    = ENV['DB_BACKUP_MYSQL'] || '/usr/local/mysql/bin/mysql'
USER     = ENV['DB_BACKUP_USER'] || 'root'

Given /^the database backup_test exists$/ do
  test_sql_file  = File.join(File.dirname(__FILE__),'..','..','setup_test.sql')
  command = "#{MYSQL} -u#{USER} < #{test_sql_file}"
  stdout,stderr,status = Open3.capture3(command)
  unless status.success?
    raise "Problem running #{command}, stderr was:\n#{stderr}"
  end

end
```

The first two lines set up some defaults for how we're going to run mysql to load the database. In our case, we set the location of the mysql executable and the username we'd like to use when loading the data. We allow this to be overridden via an environment variable so that other developers who might have a different setup can run these tests. This shows some of the complication involved in doing true end-to-end tests. These environment variables should *definitely* be documented in our README file.

Next, we need to define the steps for our two "Thens" (that the backup file should, or should not, be gzipped). To do that, we'll construct the filename we expect db_backup.rb to output, using a .gzip extension or not, depending on the test. Since the filename will contain the current date, we'll need to construct the filename dynamically:

tolerate_gracefully/db_backup/features/step_definitions/system_test_steps.rb
```
def expected_filename
  now = Time.now
  sprintf("backup_test-%4d-%02d-%02d.sql",now.year,now.month,now.day)
end
```

```
Then /^the backup file should be gzipped$/ do
  step %(a file named "#{expected_filename}.gz" should exist)
end

Then /^the backup file should NOT be gzipped$/ do
  now = Time.now
  step %(a file named "#{expected_filename}" should exist)
end
```

Using the expected_filename method, we can defer to an Aruba-provided step a file named "xxx" should exist to perform the actual check. Now, when we run our tests, everything passes:

```
$ rake features
Feature: Do a complete system test

  Scenario: End-to-end test using a real database
    Given the database backup_test exists
    When I successfully run `db_backup.rb --force -u root backup_test`
    Then the backup file should be gzipped

  Scenario: End-to-end test using a real database, skipping gzip
    Given the database backup_test exists
    When I successfully run `db_backup.rb --force -u root --no-gzip backup_test`
    Then the backup file should NOT be gzipped

2 scenarios (2 passed)
6 steps (6 passed)
0m0.745s
```

It's good to be able to actually run our apps in the exact way a user would; however, it's not always going to be possible. Even for something as simple to set up as db_backup.rb, it's still difficult. Another developer will need to set up a MySQL database and make sure it's running, just to run our tests.

You may be creating an even more complex app that interacts with other systems. If you can't figure out a way to set up a good test environment with Aruba, it's still worth writing the automated test but keeping it out of the list of features you run regularly. (Setting this up can be done with the "tags" feature of Cucumber. The documentation[2] should be able to get you started.) Whenever you are ready to release a new version or just want to do a full system test, you can manually set up the proper conditions and have Cucumber run your system test. It's not ideal, but it works for complex apps that can't easily be tested like this.

2. https://github.com/cucumber/cucumber/wiki/Tags

Everything we've talked about up to now has focused on testing our command-line apps by running them the way a user would. This gives us a clear picture of how an app is supposed to work, and we could even use our Cucumber features as supplemental documentation! Where the use of Cucumber starts to break down is when we need to test edge cases. Consider `todo`. What if the to-do list isn't formatted correctly? What if the task list file isn't writable when we add a new task? What would the app do? What *should* it do?

We went through a fair amount of effort faking out our home directory in testing `todo` (not to mention the effort we went to in order to test `db_backup.rb`!). It's going to be even more difficult to set up the conditions that simulate every edge case; it may not even be possible. We still want to test these edge cases, but we aren't going to be able to do it at the acceptance test level using Cucumber. We need to break down our code into smaller, testable units. When we do that, we can test bits of logic in isolation and can more easily simulate some strange error conditions simply in code. These types of tests are called *unit tests*.

## 8.2 Testing in Isolation with Unit Tests

In addition to allowing greater flexibility in simulating edge cases, unit tests have two other advantages: they run very quickly, since they don't require any setup outside of Ruby (such as files, databases, and so on), and they force us to organize our code into small, testable units. Faster tests are good, since we can more quickly see the health of our app and more quickly and frequently run tests when writing new features and fixing bugs. Having small testable units is good, too, because it means our app will be easier to understand and maintain; instead of a big long block of code, we'll have small units that do simple things, all glued together to form the app.

To run unit tests, we'll need to break our code into units that can be tested. Since we have a few tests in place via Cucumber, we'll have some assurance that the code changes we're about to make didn't break anything. This process is called *refactoring* and is very difficult to do without a suite of tests. Let's focus on `todo` and extract code out of `bin/todo` and into some files in `lib` that `bin/todo` can include. Our soon-to-be-created unit tests can then include these files for testing without having to execute `todo`.

### Extracting Units from Existing Code

The source code for `todo` is organized around the GLI `command` methods and blocks. Each `action` block contains the core logic of our to-do app. This is the

logic we need to extract so our unit tests can execute it. Here's the action block for the new command:

tolerate_gracefully/todo/bin/todo

```
c.action do |global_options,options,task_names|
  File.open(global_options[:filename],'a+') do |todo_file|
    if task_names.empty?
      puts "Reading new tasks from stdin..."
      task_names = STDIN.readlines.map { |a| a.chomp }
    end
    tasks = 0
    task_names.each do |task|
      todo_file.puts [task,Time.now].join(',')
      tasks += 1
    end
    if tasks == 0
      raise "You must provide tasks on the command-line or standard input"
    end
  end
end
```

Since Ruby is an object-oriented language, it makes sense to put the code currently in the action block into a class or module and then use it inside the action block. Ultimately, we'd want a class named Task that handled all things task-related, but let's take things one step at a time. All we need is to move the code out of our executable, so we'll create a module named Todo and create a method called new_task inside it. new_task will be a straight copy of the code from our action block:

tolerate_gracefully/todo_unit_tests/lib/todo/todo.rb

```
module Todo
  def new_task(filename,task_names)
    File.open(filename,'a+') do |todo_file|
      tasks = 0
      task_names.each do |task|
        todo_file.puts [task,Time.now].join(',')
        tasks += 1
      end
      if tasks == 0
        raise "You must provide tasks on the command-line or standard input"
      end
    end
  end
end
```

In Ruby, a module can be used for many things, but here, we're using it as a place where code can live that isn't naturally part of a class. Later in the book, we'll make a proper class and have a better OO design for todo, but for now, a module will accomplish our goal of unit testing this code.

Next, we need to remove this code from bin/todo, require lib/todo/todo.rb, and include the Todo module so we can access our new_task method.

tolerate_gracefully/todo_unit_tests/bin/todo

```
➤ $LOAD_PATH << File.expand_path(File.dirname(__FILE__) + '/../lib')
  require 'rubygems'
  require 'gli'
  require 'todo_version'
➤ require 'todo/todo.rb'

➤ include Todo
```

The first highlighted line isn't new (GLI included it for us when we first generated our project), but it's worth pointing out because this is how bin/todo will be able to access our newly extracted code that lives in lib/todo/task.rb. All we're doing is placing our lib directory into the load path (accessible in Ruby via $LOAD_PATH).

The next highlighted line requires our code, while the following includes into the current context. This means that any method defined in the Todo module is now available directly for use in our code. Now we can use it in the action block for the new command:

tolerate_gracefully/todo_unit_tests/bin/todo

```
c.action do |global_options,options,task_names|
  if task_names.empty?
    puts "Reading new tasks from stdin..."
    task_names = STDIN.readlines.map { |a| a.chomp }
  end
➤ new_task(global_options[:filename],task_names)
```

When we run our Cucumber tests via rake features, we'll see that all the tests are still green (and thus still passing). This means that our app is still working in light of this fairly major change in its structure. Now, we can start testing this code.

### Setting Up Our Environment to Run Unit Tests

GLI gave us the files and Rake tasks we need to start running unit tests, but let's go over the basics so you can set up unit testing for any project. We'll need to do two things: configure our Rakefile to run unit tests and create one or more files that contain our unit tests.

Setting it up in our Rakefile is simple, since rake includes the class Rake::TestTask, which sets up a rake task for running unit tests. We simply require the right module and set it up like so:

tolerate_gracefully/todo_unit_tests/Rakefile
```
require 'rake/testtask'
Rake::TestTask.new do |t|
  t.libs << "test"
  t.test_files = FileList['test/tc_*.rb']
end
```

We can now run unit tests in any file in the directory test that starts with the prefix tc_ by typing rake test.

Next, we need to create at least one file to run. All we have to do is create a file that contains a class that extends Test::Unit::TestCase and has at least one method that starts with the prefix test_. When we do this, rake test will run each test_ method as a unit test. Let's see it in action:

```
require 'test/unit'

class TaskTest < Test::Unit::TestCase
  def test_that_passes
    assert true
  end

  def test_that_fails
    assert false
  end
end
```

Now, we run our two tests and see what happens:

```
$ rake test
Started
F.
Finished in 0.003429 seconds.

  1) Failure:
test_that_fails(TaskTest)
    ./test/tc_task.rb:36:in `test_that_fails'
<false> is not true.

2 tests, 2 assertions, 1 failures, 0 errors
rake aborted!

(See full trace by running task with --trace)
```

One test passed, and the other failed. This gives us an idea of what to expect when writing and running unit tests. Let's remove these fake tests and write a real test for the code we just extracted.

### Writing Unit Tests

Unlike our acceptance tests, our unit tests should not interact with the outside world; we want to test our code in complete isolation. This is the only way we can be sure that every aspect of the tests we write can be controlled. For our purposes here, it means we have to figure out how to prevent the call to File.open from opening an actual file on the filesystem.

What we'll do is *stub* the open call. Stubbing is a way to change the behavior of a method temporarily so that it behaves in a predictable way as part of a unit test. The open source library Mocha[3] allows us to do just that. When we include it in our tests, Mocha adds a stubs method to every single object in Ruby (including the class object File) that allows us to replace the default behavior of any method with new behavior.

To see this in action, let's test that add_task raises an exception when no tasks are passed in. Since File.open takes a block and we need that block to execute (that's where all the code in add_task is), we'll use the method yields, provided by Mocha, to stub open so that it simply executes the block it was given. We'll then pass in an empty array to new_task and use the method assert_raises, provided by Test::Unit, to assert that new_task raises an exception.

tolerate_gracefully/todo_unit_tests/test/tc_task.rb

```
include Todo

def test_raises_error_when_no_tasks
  File.stubs(:open).yields("")

  ex = assert_raises RuntimeError do
    new_task("foo.txt",[])
  end
  expected = "You must provide tasks on the command-line or standard input"
  assert_equal expected, ex.message
end
```

Notice how we also include the Todo module here so that our tests have access to the method we're testing. Back to the code, the first line of our test method does the stubbing. This tells File that when someone calls open to yield the empty string to the block given to open, instead of doing what open normally does. In effect, the variable todo_file in new_task will be set to the empty string instead of a File. This isn't a problem, since the path through the code we're simulating won't call any methods on it. Instead, new_task will realize that no tasks were added and raise an exception.

3. http://mocha.rubyforge.org/

assert_raises verifies that the code inside the block given to it raises a RuntimeError. It also returns the instance of the exception that was thrown. We then make sure that the message of that exception matches the message we expect.

Since we've replaced open with our stub during the test, we also need to restore it back to normal once our test has run. Test::Unit will run the method teardown in our test class after each run (even if the test itself fails), so we can put this code there, using Mocha's unstub method to remove any stubs we've created.

tolerate_gracefully/todo_unit_tests/test/tc_task.rb

```
def teardown
  File.unstub(:open)
end
```

*Now* we can run our test and see what happens:

```
$ rake test
Started
.
Finished in 0.000577 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

Everything passed! Now that we know how to fake out File.open, we can write a complete set of tests for our new_task method. There are two major cases we need to cover: the normal execution of adding a new task and the case where we don't have permissions to read the file.

To test the normal case, we need to verify that the tasks we pass to new_task are written to the file. Since we don't want to actually write to the file, we'll use our newfound ability to stub the File.open method to capture what new_task writes out. We'll do this by yielding an instance of StringIO to the block given to File.open. StringIO looks and acts just like a real file, but it saves its data internally and not on the filesystem. We can pull that data out and examine it. That's exactly what we need to do, so let's see the test:

tolerate_gracefully/todo_unit_tests/test/tc_task.rb

```
def test_proper_working
  string_io = StringIO.new
  File.stubs(:open).yields(string_io)

  new_task("foo.txt",["This is a task"])

  assert_match /^This is a task,/,string_io.string
end
```

When we call add_task now, the file that gets yielded will be our variable string_io. When add_task calls puts on it, it saves the string internally, which we can then

examine via the string method on string_io. We assert that that string matches a regular expression containing our task name (we use a regexp here because the current date/time will also be written out).

Let's run this test and see what happens:

```
$ rake test
Started
..
Finished in 0.001007 seconds.

2 tests, 3 assertions, 0 failures, 0 errors
```

This test also passed. To prove that File.open did not create a file, we'll see if foo.txt is in our current directory:

```
$ ls foo.txt
ls: foo.txt: No such file or directory
```

The last case is the trickiest one and is the reason we've started writing unit tests; we want to make sure add_task gives a reasonable error message when the task list file cannot be written to. If this were to happen in real life, File.open would throw an Errno::EPERM exception. This exception gets its name from the C standard library's constant for a lack of permissions. We'll stub File.open to throw that error. We don't want add_task to throw that exception, however. We want it to throw a RuntimeError, and we want that exception to have a useful message that includes the message from the underlying exception. Let's see the test:

tolerate_gracefully/todo_unit_tests/test/tc_task.rb

```
def test_cannot_open_file
  ex_msg = "Operation not permitted"
  File.stubs(:open).raises(Errno::EPERM.new(ex_msg))
  ex = assert_raises RuntimeError do
    new_task("foo.txt",["This is a task"])
  end
  assert_match /^Couldn't open foo.txt for appending: #{ex_msg}/,ex.message
end
```

Now, when we run our unit test, it fails:

```
$ rake test
Started
F..
Finished in 0.008249 seconds.

  1) Failure:
test_error(TaskTest)
  [./test/tc_task.rb:44:in `test_error'
  mocha/integration/test_unit/ruby_version_186_and_above.rb:22:in `__send__'
```

```
  mocha/integration/test_unit/ruby_version_186_and_above.rb:22:in `run']:
<RuntimeError> exception expected but was
Class: <Errno::EPERM>
Message: <"Operation not permitted">
---Backtrace---
lib/mocha/exception_raiser.rb:12:in `evaluate'
lib/mocha/return_values.rb:20:in `next'
lib/mocha/expectation.rb:472:in `invoke'
lib/mocha/mock.rb:157:in `method_missing'
lib/mocha/class_method.rb:46:in `open'
./lib/todo/task.rb:4:in `new_task'
./test/tc_task.rb:45:in `test_error'
./test/tc_task.rb:44:in `test_error'
mocha/integration/test_unit/ruby_version_186_and_above.rb:22:in `__send__'
mocha/integration/test_unit/ruby_version_186_and_above.rb:22:in `run'
---------------

3 tests, 4 assertions, 1 failures, 0 errors
rake aborted!
```

We get a big, nasty backtrace, and we see that instead of getting a RuntimeError, we got an Errno::EPERM. This isn't surprising, since our test forced that to happen. What's missing here is the code to translate that exception into a RuntimeError. We'll fix it by catching SystemCallError (which is the superclass of all Errno::-style errors) and throwing a RuntimeError with a more helpful message.

tolerate_gracefully/todo_unit_tests/lib/todo/todo.rb

```
def new_task(filename,task_names)
  File.open(filename,'a+') do |todo_file|
    tasks = 0
    task_names.each do |task|
      todo_file.puts [task,Time.now].join(',')
      tasks += 1
    end
    if tasks == 0
      raise "You must provide tasks on the command-line or standard input"
    end
  end
rescue SystemCallError => ex
  raise RuntimeError,"Couldn't open #{filename} for appending: #{ex.message}"
end
```

Now, our test passes with flying colors:

```
$ rake test
Started
...
Finished in 0.00192 seconds.

3 tests, 5 assertions, 0 failures, 0 errors
```

We've covered all the paths through this method. To continue testing `todo` with unit tests, we'll continue extracting code into testable units and writing tests. The ability to stub out methods is very powerful and enables us to get very good test coverage. This is one of the benefits of working with a dynamic language like Ruby.

## 8.3 A Word About Test-Driven Development

We've built our apps a bit backward from the accepted practice in the Ruby community. You *should* be writing your tests first, using them to drive the development of features. We didn't do that; we started with code and added tests afterward. This was done purely to make it easier to learn concepts about command-line app development. We needed to know *what* to test before learning *how* to test. To be clear, we are *not* endorsing "test-last development."

To write command-line apps using Test-Driven Development (TDD), you can apply the same principles we've learned here, but just start with the tests instead of the code. (See *Kent Beck's Book on TDD* [Bec02].) The simplest thing to do is to start using Cucumber and Aruba to identify the user interface and user-facing features of your app. Write one scenario at a time, get that working, and move on to a new scenario. Repeat this process until you have the basic "happy paths" through your app working. Simulate a few error cases if you can, but at that point, you'll want to turn your attention to unit tests, extracting your mostly working code into testable units and putting them through the ringer to iron out all the edge cases.

## 8.4 Moving On

We only scratched the surface of the art of software testing, but we went through a whirlwind tour of everything you'll need to get started for testing command-line apps. We saw some real challenges with testing our apps, as well as several techniques to deal with them. By manipulating the environment, setting up test-specific infrastructure, and mocking system calls, we can simulate almost anything that might happen when our app runs.

Toward the end, when we learned about unit testing, we talked briefly about refactoring. Refactoring is difficult without tests, but with a good suite of tests, we can safely change the internal design of our code. We got a taste of that when we extracted our business logic out of `bin/todo` and put it into `lib/todo/task.rb` so we could unit test it. In the next chapter, we'll learn some patterns and techniques for organizing our code so that it's easy to maintain, test, and enhance.