Proposal for

"Creating Awesome Command Line Applications with Ruby"

by David Copeland / `davidcopeland@naildrivin5.com`

This book is about how (and why) to create robust, maintainable, and easy-to-use command line applications using Ruby. A rough draft of it was written during the most recent PragProWriMo, and clocks in at about 32,000 words.

The book first establishes the continued need for command line applications, and makes the case that these applications should be just as polished and robust as any other application. The book makes a short case for Ruby as the perfect language, at the perfect time, to allow anyone to make awesome command line applications very simply.

I then walk the reader through the creation of three different, but real-world, command line applications, each demonstrating a series of best-practices (anything from "exit with nonzero if you encounter an error" to "provide a config file for user-specific defaults"). These best-practices are all motivated by a real-life problem in the applications being developed. The results show a cleaner and more maintainable application

Theory and practice go hand-in-hand: the reader is left with a comprehensive theory on command line application design, as well as a list of concrete practices and tools to make it happen.

Who's the audience? What's the market?

The audience of this book is, potentially, any programmer or system administrator. Command line applications are so prolific that anyone working as a developer or sysadmin will be able to make use of this book.

The primary audience will be Ruby developers and sysadmins familiar with Ruby. There have been over 170,000 gem downloads across several Ruby command-line gems; there's a lot of command-line development going on!

Further, since this book isn't about a wholesale change in technical architecture (as a book about Rails might be), it's going to be accessible to more than just Ruby programmers; Java and C++ programmers have just as strong a need for robust command-line tools as Ruby developers.

What makes this book different? Who's the competition?

I had a hard time finding *any* book on this topic, Ruby or not. There are chapters on command line applications in some UNIX books and some `bash` books, but nothing this comprehensive. I found that surprising.

In addition, this book distinguishes itself by a mix of theory, practice, **and** opinion. I tell the readers that their application's output should be greppable, and explain why. I tell them when it's OK to break this rule, and why. I won't just talk about how to use `OptionParser`, but I'll explain why you should

provide both the short *and* long forms of arguments – I doubt you'll see another book on command line usability!

Finally, it's a book about writing applications in Ruby that has nothing to do with Rails; Ruby's not just for writing web apps.

What will readers get out of this book?

On first read, the user will experience a tour of all that is possible, how to go about doing it, and many best-practices. After that, readers can flip to specific sections to see the best practices along with clear examples that they can apply to their work. Readers will start to think of command-line applications as being first-class citizens *and* they'll be given the direction and instruction on how to make that happen.

Why should anyone get excited about this book?

- There isn't one like it right now.
- It shows how productive Ruby can be outside of Rails.
- Java and C++ developers can use this right away without convincing their boss to "switch" to anything.

How would I promote this book?

- I will speak at any conference or user group and preach the gospel of this book. I've already done so a few times, and I think this is a great way to spread ideas.
- I could port high-profile Ruby apps using techniques described in this book and blog about it; compare and contrast what I did in a series of blog posts and reference the book.
- The information in this book scales extremely well: I could do a short article in PragProg Magazine, or a series of 3-5 minute screencasts demonstrating the topics in here.

OK, so what's the book look like from a high-level?

The majority of this book exists as a first-draft written during PragProWriMo. It's currently 38,175 words (including code). A few of the chapters below don't exist, so I'd expect the book to be 200-250 pages. The "meat" of the book is section 4, "How?".

1. Why
   (a) Why are command line apps important?
   (b) Why should we make them awesome?

   (c) Why Ruby?

2. What makes an awesome command line app?
3. Ruby, Object-orientation, and UNIX for the impatient
4. How?
   (a) Making a simple automation script
   (b) Making a complex "command-suite style" application
   (c) Making a command line application with a rich user experience
5. Other Tools & Techniques
   (a) Working on Windows
   (b) Alternative Distribution (apt, yum, brew, etc.)
   (c) Application Design/Implementation Patterns
   (d) Testing
   (e) Distributing your application as open source
6. Best-Practices Reference

Who is the author, again?

I'm David Copeland. I've got 15 years of software development experience and I love to write. My first job was programming C, but I was quickly tasked with automating our build system in Perl. I've always been a command line guy, even though I write primarily Java code at work. I actively write Ruby code at home, and introduced it to my current employer in the form of command line automation scripts and automated testing using cucumber.

I also appreciate (and demand) polish and usability from my tools and see no reason the command line can't be just as perfect as the iPhone. I've seen firsthand the ill-effects of crappy `bash` scripts and the wonders of polished command line interfaces. I also created a command-suite style command line parser and mini-framework called GLI that I've used many times to create Ruby command line applications quickly and easily that meet my standard of usability.

As I said, I love to write, and I want to write well; I want to work with someone that can help me 'level up' in my writing skills while working on a compelling and, hopefully great-selling, book.

How about a sample chapter?

A sample chapter follows this section. This chapter is part of proposed section 4.a (see outline, above), in which the reader is creating a basic application to back up their database. The reader has started from something, well, bad, and, together, we've improved it a little bit. This chapter picks up there with some additional improvements.

Sample Chapter - Improving our Backup Script

In the previous chapter we took a simple-but-flawed backup script and turned it into a much more well-behaved Ruby application. We ensured no duplicate filenames were created, added some messaging during the backup process to let us know of problems, and used our exit codes to make the script play well with others. But now we have 23 lines of ugly and repetitive code. This is going to make it hard for us to add some new features later on, so what can we do?

Removing Repetition

If we extract a few things from our repetitive code, we can reduce the line count *and* complexity significantly:

```
#!/usr/bin/ruby
mysqldump = 'mysqldump'

COMMANDS = {
  "foo" => "foo_db",
  "bar" => "-uspecial -pspecialp@ss bar_db",
}

now = Time.now
date_part = "#{now.year}-#{now.month}-#{now.day}"
failures = 0
COMMANDS.each do |database_name,command|
  cmd = "{mysqldump} #{command} > #{database_name}_#{date_part}.sql"
  puts "Running #{cmd}"
  if system(cmd)
    puts 'Backup of #{database_name} successful'
  else
    puts "Backup of #{database_name} failed $!"
    failures += 1
  end
end
exit failures
```

This is quite a bit cleaner and it also exemplifies why we want to use a high-level language. Ruby's `Hash` literals make it very easy to create an on-the-fly data structure that we can iterate over. In this case, we use the database name as the key (which we can also use for messaging), and map it to the command-line options needed when calling `mysqldump`. We can now see all the databases we're going to backup, but also the differences needed in accessing each of them. All of the boilerplate (calling `mysqldump`, redirecting to a nicely named file) is handled inside the block.

Notice further that we've done a few other things to make this code clean. `COMMANDS`, our hash of databases to back up, is a constant. There's no technical

reason it has to be a constant in such a small script, but we're using the features of Ruby to communicate to anyone reading this code: `COMMANDS` is not to be changed while the program's running. If our program grows in size, and another programmer adds code to modify `COMMANDS` in the future, she'll get a warning and realize her mistake. This keeps things more maintainable

We've also slightly changed how we use our exit codes. Before, we'd exit with -1 if anything went wrong. Now we exit with the number of failures we encountered. The effect is still the same from the perspective of another program that might call ours (nonzero means an error occurred), however we've added some new information in here in case anyone wants it (and simplified our code by not using a boolean flag).

There's still a bit of a problem, however. The output of this command is littered with logging statements *and* errors. `ls` certainly never outputs "Files listed successfully!" after we call it, so why should our application be so chatty? Further, we have no way to distinguish informational messaging from error messaging.

Best Practice: Know where, and when, to send messages

UNIX established a convention regarding program output that is extremely useful. Every process has, by convention, two output streams. One is called the "standard output" and is intended for the output of the command. For example, `ls` writes the list of files to the standard output and `grep` prints the matches it finds in text files to the standard output.

For error messages, the second stream is used, and it's called "standard error". For example, if you were to run the command `ls foo` in a directory with no file named "foo", you would get the message "ls: foo: No such file or directory", and it would be printed to the standard error.

UNIX shells (such as `bash`) know about these two output streams, and allow you to redirect them to different places if you wish. In `bash` if you execute the command `ls > output 2> error` followed by `ls foo > output2 2>error2` you can see the two streams separately:

```
$ ls > output1 2> error1
$ ls foo > output2 2> error2
$ cat output1
Rakefile
how/
intro.md
other/
what/
why/
$ cat output2
# File is empty
$ cat error1
# File is empty
```

```
$ cat error2
ls: foo: No such file or directory
```

The form `> filename` tells `bash` to redirect the standard output to `filename`. The form `2> filename` tells `bash` to redirect the standard error to filename. Finally, you can send them both to the same place via `> filename 2>&1`. The numbers map to UNIX file descriptors, with 1 being the file descriptor for standard output and 2 for standard error.

Users of command line programs expect messages to go to one of these two streams based on this convention. As we'll see, this allows them to do something intelligent with the output.

Since our script doesn't process any text or generate any particular output, we can continue using its standard output as an informational log. Of course, we now want to change where we send the error messages:

```
$stderr.puts "Backup of #{database_name} failed $!"
```

We just change that line to use the Ruby built-in variable `$stderr`. This means that that message will now appear on the standard error and not the standard output (Our previous call to `puts` is implicitly called on `$stdout`, which is the standard output; the `puts` method that comes from the `Kernel` module uses the standard output by default).

Since `cron` knows about this UNIX convention, we can take advantage of this. We'll change our `crontab` entry to redirect only the standard *output* to a file, leaving `cron` to receive anything printed to the standard error:

```
30  23  *  *  *  /home/davec/backups.rb >> /home/davec/backups.log
```

The log messages will now be appended (the `>>` says to redirect standard error to the given file, *appending* to what's there, rather than overwriting the file) to `/home/davec/backups.log`, while standard error will be available to cron. Since cron sends an email to the user running a scheduled command *only* when that command outputs something, we'll get an email from cron *only* when our script encounters an error!

Adding features

Our script has already come a long way, however it still has a pretty serious problem: it will eventually (and quickly) fill the disk with backups. We now need to make a decision about how to handle this. Let's revisit why we have this script at all: we have test data that we wish to backup in case of database failure. So, we don't need every single database backup for every day of development. However, we probably want snapshots of the database at the end of each of our iterations (since we're a startup, we use an agile methodology, and work in iterations; ours last three weeks each).

So, talking with our QA people, we decide on the following retention policy:

- At most 5 days of backups per database, as recent as possible
- One backup per database for each iteration
- Compress all backups

This should save quite a bit of disk space; database dump files tend to be a series of SQL statements, and should compress nicely. Further, we'll end up with one good backup for each iteration, and the previous 5 days as well, to track development of our in-progress iteration.

Adding complex features like this is where our use of a high-level languages (and Ruby, in particular) is really going to shine. Let's sketch out how our script will look, keeping to an imperative design (new lines are followed by a comment and a reference number):

```
#!/usr/bin/ruby
mysqldump = 'mysqldump'

iteration = ARGV[0] #1
COMMANDS = {
  "foo" => "foo_db",
  "bar" => "-uspecial -pspecialp@ss bar_db",
}

now = Time.now
date_part = "#{now.year}-#{now.month}-#{now.day}"
failures = 0
COMMANDS.each do |database_name,command|
  filename = "#{database_name}_#{date_part}.sql" # 2
  unless iteration.nil? # 3
      filename = "#{database_name}_#{iteration}_#{date_part}.sql"
  end

  cmd = "{mysqldump} #{command} > #{filename}" # 4
  puts "Running #{cmd}"
  if system(cmd)
    puts 'Backup of #{database_name} successful'
    do_gzip(filename) # 5
    do_cleanup(database_name) # 6
  else
    $stderr.puts "Backup of #{database_name} failed $!"
    failures += 1
  end
end
exit failures
```

Let's go over our changes. We've added six new bits of code:

1. Here we add a command line argument that lets us indicate that we're doing an iteration backup. We do this so we can use a different naming scheme (this will become clear in a moment)

2. Here we create a variable for the filename using the same format as before. We do this so we can override it if the user has chosen an iteration backup

3. Here we use a different naming scheme only if we're doing an iteration backup (this is why we set up the command line argument in step 1). The value might be something like our iteration number, e.g. "V2.13".

4. Now, we use the filename in our command instead of having the format directly in the command string.

5. We add a method to gzip what we just backed-up. We'll see the implementation of this method later.

6. The last piece of the puzzle is here, where we call a new method (that we also haven't yet seen) that will cleanup old files. We execute this only if our database dump succeeds, and the idea is that this method will remove enough files so that we only have 5 left.

The way we've done this is a very simple and effective technique for adding new features. We start with the assumption that the new features will be handled by some methods we have yet to write; we sketch them out in our main routine in an idealized form. After that, we implement the methods. This is a form of *functional decomposition* and is one of the most basic means of designing software. Since our script is pretty simple, it's a perfect fit here.

Best Practice: Use Functional Decomposition When Things Start Getting Complex

Although Ruby is a high-level language, it's still a general purpose language. Think of methods that *we* write as creating our own special-purpose *higher* level language. If you read the `if...else` block in our code in English, it pretty much says what it does: "if system(cmd) then do gzip and do cleanup."

Functional decomposition is a great way to make your code readable. You can either reference functions you plan to write (as we're doing here), or you can extract complex code that already exists into functions to make things cleaner. For example, we could create a method called `get_filename` that handles the logic of formatting the file name for us:

```
def get_filename(database_name,iteration)
  date_part = "#{now.year}-#{now.month}-#{now.day}"
  if iteration.nil?
    "#{database_name}_#{date_part}.sql"
  else
    "#{database_name}_#{iteration}_#{date_part}.sql"
```

```
  end
end
```

Now, our `COMMANDS.each` block looks like so:

```
COMMANDS.each do |database_name,command|
  filename = get_filename(database_name,iteration)

  cmd = "{mysqldump} #{command} > #{filename}" # 4
  puts "Running #{cmd}"
  if system(cmd)
    puts 'Backup of #{database_name} successful'
    do_gzip(filename) # 5
    do_cleanup(database_name) # 6
  else
    $stderr.puts "Backup of #{database_name} failed $!"
    failures += 1
  end
end
```

Back to our code, it's certainly getting complex enough to use functional decomposition, especially in light of our new features that implement the retention policy, however we now have a new problem. In Ruby (as in most scripting languages), the code executes top to bottom. If we want to extract our code into a method, we have to define that method somewhere in the file before we call it. This means that our source file will start off with all of our support methods, and the "meat" of what our application does can't be seen without going all the way to the bottom. We want to see the main logic of our application right when we jump into with our editor. A good way to do that is to extract our main logic into a main method, and call that at the end of the script:

```
#!/usr/bin/ruby

def main
  mysqldump = 'mysqldump'

  iteration = ARGV[0]
  COMMANDS = {
    "foo" => "foo_db",
    "bar" => "-uspecial -pspecialp@ss bar_db",
  }

  now = Time.now
  failures = 0
  COMMANDS.each do |database_name,command|
    filename = get_filename(database_name,iteration)
    cmd = "{mysqldump} #{command} > #{filename}"
```

```
    puts "Running #{cmd}"
    if system(cmd)
      puts 'Backup of #{database_name} successful'
      do_compress(filename)
      do_cleanup(database_name)
    else
      $stderr.puts "Backup of #{database_name} failed $!"
      failures += 1
    end
  end
  failures
end

def do_cleanup(database_name)
  # TBD
end

def do_gzip(filename)
  # TBD
end

failures = main
exit failures
```

We've basically kept the structure the same, but since our primary logic is now in a method, we return the number of failures, and the last line of our application simply treats this value as the exit code. Now, we are free to organize our code how we'd like. We've already created stubs for our cleanup and gzip methods.

It may seem like we've gotten seriously sidetracked with all of this code reorganization, however it's very important to give ourselves a clean place to work; think of this is doing our dishes midway through cooking a complex meal. We simply need to have a clean work surface to continue working.

Back to the matter at hand, how are we going to cleanup the extra backup files? Our strategy will be to get a list of all the files in the current directory, see which ones match the format of the database backup we just took, parse the date, and delete the oldest ones.

```
def do_cleanup(database_name)
  files = []
  Dir.entries('.').each do |file|
    files << file if file =~ /#{database_name}_\d+-\d+-\d+\.sql/
  end
  files.sort[0..-6].each do |file_to_remove|
    FileUtils.rm(file_to_remove)
  end
end
```

There's a lot going on here, but it's idiomatic Ruby for the most part. Our code builds up a list of files matching a given filename pattern (this pattern is a *regular expression*. `\d+` means "match any number of digits", so our expression means a filename "starting with our database name, followed by an underscore, followed by any number of digits, followed by a dash, followed by any number of digits, followed by another dash, and followed by more digits, ending in a `.sql`." Whew!).

Once we have that list, we sort it. Since we've chosen a filename format that, when sorted lexicographically, happens to also sort by date, we end up with an array of filenames ordered by date, which is exactly what we need. We then use Ruby's flexible (and useful) array slicing syntax, `[0..-6]` (which means, produce an array that has all items in it, save for the last 5), and iterate over each one.

We then use `FileUtils` to delete the files. In other words, we find all files but the 5 newest, and delete them; exactly our retention policy.

`FileUtils` is incredibly useful, especially if you are used to using UNIX commands in `bash` scripts. FileUtils exposes methods that look just like their UNIX counterparts. Here, we use the method `rm`, which acts just like the UNIX command of the same name. There's also an `rm_rf`, which acts just like `rm -rf` on a UNIX command line. FileUtils is part of any Ruby installation and can be accessed by adding `require 'fileutils'` anywhere in your Ruby file. If you are doing a lot of calls to `FileUtils`, you can also add `include FileUtils` and access the methods directly, without the `FileUtils.` prefix.

Finally, we need to implement gzip. This requires calling the external `gzip` command installed on most UNIX boxes, so it'll look similar to how we execute `mysqldump`:

```
gzip = "gzip"
def do_gzip(filename)
  command = "#{gzip} -9 #{filename}"
  if system(command)
    puts("Compressed #{filename}")
    true
  else
    $stderr.puts("Problem executing '#{command}': $!")
    false
  end
end
```

We extract the location of `gzip` into a variable (so we can modify it later if we need to) and execute our command pretty much the same way.

This code looks exactly the same in structure as the code we use to call `mysqldump`, so, while it's all fresh in our minds, let's extract the duplication into a new method:

```
def execute_command(command,description)
  puts "Running #{command}"
  if system(command)
    puts "#{description} successful"
    true
  else
    $stderr.puts("Problem executing '#{command}': $!")
    false
  end
end

def do_gzip(filename)
  execute_command("#{gzip} -9 #{filename}","Compression")
end

def do_backup(filename,command)
  execute_command("{mysqldump} #{command} > #{filename}",
                  "Backup of #{database_name}")
end
```

The `execute_command` method now handles all the logic of executing an external command, checking for success, logging the result and providing error messaging. We use the `description` variable to make sure our log message includes the human-readable explanation of what we were trying to do. This turns our `do_gzip` and `do_backup` methods into one-liners. We *could* eliminate them entirely at this point, however it might be handy to have them around, so we'll keep them for now.

We now have a fairly robust script that doesn't fill the disk, behaves reasonably well in the face of failure, and is easily comprehensible and maintainable. But, there's a lot more we could do.

In the next chapter, we'll add some flexibility to our program so it can be used more easily on other systems. We'll provide a richer command-line interface using `OptionParser` to expose this flexibility to the user while making our script easier to use and easier to debug.