Everyday Scripting with Ruby

For Teams, Testers, and You

Chapter 7

The Churn Project: Writing Scripts without Fuss

Scripting can be straightforward or horrible. When it's horrible, it feels like the script is actively fighting you and that every try at making something better makes something else worse. The way to make it straightforward is to proceed in tiny, tested steps. In this chapter, I'll show you how to do that.

"Straightforward" doesn't mean "error-free." Expect to make mistakes all the time; the trick is to recover from them smoothly and quickly. This chapter will demonstrate that by showing how I handle two blunders of the sort that often lead to a quagmire.

One warning: in this chapter, I'm going to explain my thinking as I write a script. It takes a lot longer to explain thoughts than to have them. Don't let all the words in this chapter fool you into thinking that scripting requires agonizing over every decision. Instead, strive to make decisions crisply. If you can't decide which of two possibilities is better, it probably doesn't matter which you pick. If you're wrong, just recover and move on.

7.1 The Project

If you ask a programmer what she's working on, she might say "auditing" or "the persistence layer." Systems are usually divided into named subsystems with boundaries that are more or less clear. The source code for different subsystems is usually stored in different folders.

```
prompt> svn log --revision 'HEAD:{2005-07-30}' svn://rubyforge.org/var/ ↩
svn/churn-demo/inventory
------------------------------------------------------------------------
r2 | marick | 2005-08-07 14:26:21 -0500 (Mon, 07 Aug 2005) | 1 line

added code to handle merger
------------------------------------------------------------------------
r1 | marick | 2005-08-07 14:21:47 -0500 (Mon, 07 Aug 2005) | 1 line

first touches
No commit for revision 0.
------------------------------------------------------------------------
```

Figure 7.1: Changes to a Subsystem

```
prompt> ruby churn.rb
Changes since 2005-08-05:
         audit ********* (45)
   fulfillment **** (19)
   persistence *** (17)
            ui *** (15)
          util * (3)
     inventory * (2)
```

Figure 7.2: Output From This Chapter's Program

When deciding where to concentrate effort, a tester might want to know which subsystems have changed the most. That information is available from the project's version control system. My favorite is called Subversion. Figure 7.1 shows one way of asking Subversion about a fake project I've set up. If you have Subversion on your system and are on the Internet, you can type the same line to get similar information. Subversion, like the best things in life, is free. You can find it at http://subversion.tigris.org. You don't need it to work on this project, though.

Subversion's output shows that the "inventory" subsystem has changed twice since July 30. That's a pretty ugly way to get the information, though, and it shows you only one subsystem at a time. In this chapter, we'll write a script that asks the same question of all the subsystems in a project, producing output like that of Figure 7.2.

The output isn't fancy, but there's nothing wrong with simple and functional. Notice the script somehow knows what subsystems there are (the six listed) and from when it should start counting changes (the date one working month before the script is run). It might be nice to allow those defaults to be overridden, but we won't bother for this script. (We will for later ones.)

7.2 Building a Solution

A technique that often works well is one I call *scripting by assumption*.[1] The trick is to start writing the script by *assuming* that Ruby provides all the methods you need. Here's some starting code I wrote for churn, all the while assuming that everything hard would be done for me:

scripting by assumption

Download churn/snapshots/churn.v1.rb

```
❶ if $0 == __FILE__
❷   subsystem_names = ['audit', 'fulfillment', 'persistence',
                        'ui', 'util', 'inventory']
❸   start_date = month_before(Time.now)

❹   puts header(start_date)
    subsystem_names.each do | name |
❺     puts subsystem_line(name, change_count_for(name))
    end
  end
```

I'll explain these lines over the next several pages. Unless you *like* flipping pages back and forth, you may find it more convenient to look at the source online. The callout symbols (like ❹) are included in the file. They're shown as *end-of-line comments* like this:

end-of-line comments

```
puts header(start_date)    #(4)
```

When it sees the comment character `#`, Ruby ignores everything from there until the end of the line.

Our script must produce two things: a header string containing not much more than the starting date of the changes, and a series of strings, one for each subsystem, that contains a name, a change count, and asterisks for a simple visual representation of the change count. The asterisks make the output look something like a histogram tipped on its side.

1. That's not a name I made up, but I can't find who said it first. Abelson and Sussman write of "programming by wishful thinking" in *Structure and Interpretation of Computer Programs* [ASS84]. It's the same idea, but I like "assumption" a bit better.

At ❹, I assume Ruby comes with a method, header, that produces a correctly formatted header string. When writing that line, I came to a fork in the road. I could have the script tell header what date to print, or I could have it figure it out (by calculating the date one month before the moment the script runs). The choice comes down to this:

```
puts header(start_date)
```

or this:

```
puts header
```

I chose the first because the second implies that the header string is always the same. It's not: it varies, so it seems sensible to make the cause of variation explicit. When I wrote that line, I didn't know what start_date really was. I assumed it'd become obvious later.

At ❺, I assume a method, subsystem_line, that returns a string ready to print. The contents of that string will vary depending on the subsystem's name and count of changes. Should the count of changes be given to subsystem_line directly or indirectly?

- "Directly" means that the script will get a subsystem's change count from Subversion and hand it to subsystem_line. In that case, subsystem_line's definition would start like this:

  ```
  def subsystem_line(name, change_count)
  ```

- "Indirectly" means subsystem_line will itself ask Subversion for the change count. To do so, it would need to know the starting date, so its definition would start like this:

  ```
  def subsystem_line(name, start_date)
  ```

I chose the direct approach because it makes it more obvious what's in the string that subsystem_line will create. It also follows a guideline called *separation of concerns*. In the indirect case, subsystem_line has two concerns: how to format a string and how to communicate with Subversion. In the direct case, subsystem_name is only about formatting, and some other method is about Subversion.

separation of concerns

Since subsystem_line has to be called for each subsystem, it makes sense to stash all the subsystem names in an array and iterate over them with each. The array is created and named at ❷. Since a project's list of subsystems will rarely change, it makes sense to "hard-code" it.

I'm assuming everyone always wants to know the number of changes in the last working month, so start_date is defined that way at ❸. I could have defined it as an argumentless method last_month, but what I've got seems to read more clearly: "the starting date is the month before right now." (When the Time object is sent the message now, it returns an object that represents the current instant of time.) And, as you'll learn in Section 7.2, *Test-driving month_before*, passing in a date also makes test-driven scripting easier.

I created a variable start_date to name the day from which to start looking for changes, but when I needed a change count to give to subsystem_line, I passed it along directly:

```
puts subsystem_line(name, change_count_for(name))
```

I could have written this:

```
change_count = change_count_for(name)
puts subsystem_line(name, change_count)
```

Why didn't I? There are two reasons for adding a variable to a script. The first is that you're using an object more than once and you're either unable or unwilling to create it twice. That doesn't apply here. The other is that the variable helps someone understand the script. My main reason for creating start_date was that I could put it next to subsystem_names at the beginning of the script. All the data the script works with depends on the data those two variables name, so it makes sense to draw attention to that by putting them first and together. I can't see any way that creating a variable change_count would help a reader.

There's one more bit of code to mention: ❶. Because of it, the script can both be run from the command line and also be loaded into irb. (The trick was explained in Section 5.7, *Prelude to the Exercises*, on page 58.)

Test-driving month_before

It's increasingly common for programmers to build their code *test-first*: if the code doesn't do something you want, first write a test that fails because of that, and then write the minimal amount of code that passes the test. If more is needed, write the next test and then the next bit of code. Continue until the code does what you want. Along the way, clean up code whenever it starts to get messy, making sure that the cleaned-up code always passes all the tests. (The technical term for such cleanup is *refactoring*.)

test-first

refactoring

With practice, writing code with tests is faster than writing code alone (because of the time you don't have to spend hunting for bugs), and it's usually a lot more pleasant.

Download template-for-tests.rb

```
❶ require 'test/unit'
❷ require 'X'

❸ class  X < Test::Unit::TestCase

❹   def test_X
❺     assert_equal('expected', 'actual')
    end

  end
```

template-for-tests.rb

Figure 7.3: A Test Template

Ruby comes with a package called *Test::Unit* that lets you set up tests without having to write much support code. You can find a test template in Figure 7.3. Parts you'll need to fill in are marked with an *X*. *Test::Unit*

❶ The test file is run like any other script. require loads all the Ruby code that makes up Test::Unit into that running script. It's almost the same as using load in irb.

❷ Usually, there's one test file for each script file. This line loads the script under test. In this case, *'X'* will be replaced with *'churn'*. (Note that, unlike load, require doesn't need the *.rb* at the end of the filename; it can figure it out.)

❸ For the moment, let's ignore what this line means beyond saying that **class**. . . **end** serves to group the tests. See Chapter 11, *Classes Bundle Data and Methods*, beginning on page 114, for more. *'X'* names the file's group or *suite* of tests. You can pick any name you want, but it must begin with a capital letter. ChurnTests seems reasonable. *suite*

❹ When you run a test file (e.g., ruby churn-tests.rb), Test::Unit executes each method whose name begins with test_. It ignores other methods. The ones it ignores can be used as utilities by the ones it does run.

❺ Each assert_equal message compares its first argument (the expected value) to the second (the actual value produced by the code under test). If they aren't equal, it complains. (You'll see a typical complaint in a minute.)

What would a test for month_before do? A working month is 28 days. So the month before January 29 is January 1. In the script proper,

Joe Asks. . .

Why Does Ruby Have Both require and load?

require and load do almost the same thing. The important difference is that require remembers the files it's loaded and will load each only once. That behavior is useful when script A.rb uses a method in B.rb and B.rb uses a method in A.rb. If A.rb had load 'B.rb' and B.rb had load 'A.rb', then loading A.rb would load B.rb, which would load A.rb, which would load B.rb. . . .

Given require, why ever use load? Suppose you're writing some code in a file. You require it into irb and try it. Oops, it's wrong. You change the file. If you require it again, you won't get the changed version (because Ruby knows you've already loaded that file). You have to use load to get the new version.

So use require in script files and load in irb.

Time.now is used as the argument to month_before, but the test doesn't have to use a Time that represents the current instant. In fact, it *can't* use that. If it used Time.now, the actual value would change every time the test ran. What would the expected value be?

Fortunately, Time provides methods that construct any arbitrary time: local is the one that constructs Times relative to the local time zone. That means we can ask month_before to pass this test:

Download churn/snapshots/churn-tests.v1.rb

```ruby
def test_month_before_is_28_days
  assert_equal(Time.local(2005, 1, 1),
               month_before(Time.local(2005, 1, 29)))
end
```

Let's see that test fail.[2]

Before you can run the test, you'll need to copy it from the snapshots folder. On Windows, you do that like this:

```
prompt> copy snapshots\churn-tests.v1.rb churn-tests.rb
```

2. It's common practice to run a test even if you *know* it will fail. I thought that was a silly ritual until the first time I did it and saw no failure. I had put the test in the wrong place, so Test::Unit hadn't run it. If I hadn't tried the test first, I might have written bad code, run the tests, seen no failure, and thought I'd done well. That would have been bad.

On Unix-like systems, it's like this:

```
prompt> cp snapshots/churn-tests.v1.rb churn-tests.rb
```

All the tests assume they're testing churn.rb. Unless you've already created it, do that now by copying churn.v1.rb from the snapshots folder into the current working folder. Be sure to copy it into churn.rb.

Having prepared the test, run it like this:

```
prompt> ruby churn-tests.rb
Loaded suite churn-tests
Started
E
Finished in 0.002627 seconds.

  1) Error:
test_month_before_is_28_days(ChurnTests):
NoMethodError: undefined method ‘month_before' for #<ChurnTests:0x32f8f0>
    churn-tests.rb:9:in ‘test_month_before_is_28_days'

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

The test correctly tells us that there's no method named month_before yet. Let's define it. But where?

The test will requirechurn.rb. That means Ruby will ignore the body of the if $0 == __FILE__ check. (See Section 5.7, *Prelude to the Exercises*, on page 58.) So it should be above the **if**.

In order to see a more typical failure, let's define month_before wrong:[3]

Download churn/snapshots/churn.v2.rb

```
def month_before(a_time)
end
```

Here's the failure:

```
  1) Failure:
test_month_before_is_28_days(ChurnTests) [churn-tests.rb:9]:
<Sat Jan 01 00:00:00 CST 2005> expected but was
<nil>.

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

An empty method returns nil, which certainly isn't the Time expected. Notice that the failure message identifies both the test that failed

3. I usually don't bother running a test that fails because the method isn't defined. I define an empty method before running the test for the first time.

Joe Asks...

What's the Difference Between an Error and a Failure?

A Test::Unit test can fail in two ways. Our second test run showed a failure. A *failure* means that what an assertion asserts to be true isn't in fact true. An *error* means that something else went wrong before the assertion was tried. In our first test run, Ruby stopped the script when it discovered there was no such method as month_before and, therefore, no return value for assert_equal to compare against January 1, 2005.

Frankly, I always have to think for a minute when I'm asked which is which. Perhaps that's because I treat both cases the same. I look at the explanation of what went wrong, I go to the line number mentioned, and I fix the problem.

(test_month_before_is_28_days) and the line it failed on (line 9). The latter is useful when there's more than one assertion in a test.

Let's write the right code. To get an earlier Time, you subtract some number of seconds:

```
irb(main):001:0> now = Time.now
=> Mon Aug 29 11:42:19 CDT 2005
irb(main):002:0> now-1
=> Mon Aug 29 11:42:18 CDT 2005
```

So all we have to do is subtract 28 days of seconds:

Download churn/snapshots/churn.v3.rb

```
def month_before(a_time)
  a_time - 28 * 24 * 60 * 60
end
```

And it passes:

```
prompt> ruby churn-tests.rb
Loaded suite churn-tests
Started
.
Finished in 0.00247 seconds.

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

It's such a satisfying moment when that happens. Those frequent jolts of pleasure are what makes test-driven scripting so satisfying. You won't believe it until you try it.

Formatting Time

Now that we believe month_before works, we also believe that start_date will name the right object after this assignment:

```
start_date = month_before(Time.now)
```

So it makes sense to now write the method, header, that uses the returned Time:

Download churn/snapshots/churn.v3.rb

```
puts header(start_date)
```

Here are two possible tests:

Download churn/snapshots/churn-tests.v2.rb

```
def test_header_format
  assert_equal("Changes since 2005-08-05:",
               header(Time.local(2005, 8, 5)))
end

def test_header_format
  assert_equal("Changes since 2005-08-05:",
               header(month_before(Time.local(2005, 9, 2))))
end
```

For both, I just copied the expected output from Figure 7.2, on page 70. The difference between the two is how they generate the value given to header. One is a *direct test*: it uses Time.local to make exactly the Time it needs. The other is a *bootstrapping test*: it uses an already-tested method from the script under test to test a new method.

direct test

bootstrapping test

The two types of tests have contrasting advantages. A direct test is usually easier to understand. It's also usually easier to debug if it fails. Suppose that I later change month_before and break it. Then both the direct test_header_format and the test_month_before_is_28_days will fail. I'll have to decide which one to look at. If I look at test_header_format, I have to wonder whether the problem is in header or in month_before. That's hardly a big deal in this case, but it can get cumbersome when you have 200 tests that use month_before. It's even worse if the change to month_before *deliberately* changes its behavior (maybe I want it to return a different kind of object). Then I may have to fix all 200 tests.

The advantage of bootstrapping tests is that they're more realistic and thorough. Suppose it turns out that there's a mismatch between what month_before returns and what header expects. Perhaps month_before returns a Time and header expects a string. A bootstrapping test for header would detect that, but a direct test would not, since the test that header is to pass will use the string it expects.

A second advantage of bootstrapping tests is that they use month_before again. I made sure I tried a different kind of value in the test of test_header_format than I did in the earlier test for month_before. Since, in the previous test, both "now" and the date 28 days earlier are in the same month, this time I picked ones in different months. I have no reason to think that will find a bug, but I've found too many bugs through sheer dumb luck to use the same value twice. On the other hand, figuring out what date to hand to month_before to cause it to hand August 5 to header was more work than the direct test requires. Was it worth it?

Different people have different biases. Mine is toward bootstrapping tests, so I'll use the second version. But if I had a lot of tests to write for header, I'd make only a couple of them bootstrapping. I'd make the rest of them direct so that I didn't face the prospect of changing many tests if I ever change my mind about what month_before should do.

Everyone finds their own balance between testing directly and testing indirectly. You will too.

The only tricky part about implementing header is formatting dates. Ruby's default format (from the scripter-friendly inspect message) is something like "Mon Aug 29 12:20:00 CDT 2005". That's not the format we want. Fortunately, Time objects respond to the gracefully named strftime message. And here's an example of the result:

```
irb(main):002:0> Time.now.strftime('%Y-%m-%d')
=> "2005-08-29"
```

Each character that follows % picks out a piece of the Time and places the result in the string strftime returns. You can find the complete table of format characters either in a Ruby reference like *Programming Ruby* [TH01] or like this:

```
prompt> ri Time.strftime
```

(You can find information about ri in the sidebar on page 121.)

header can just be a more elaborate format string:

Download churn/snapshots/churn.v3.rb

```
def header(a_time)
  a_time.strftime("Changes since %Y-%m-%d:")
end
```

The test passes:

```
prompt> ruby churn-tests.rb
Loaded suite churn-tests
Started
..
Finished in 0.006264 seconds.

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

Note that the earlier test still passes. That's good to know.

Formatting Strings

subsystem_line is another method that is about formatting, so let's do that next. Here's a test:

Download churn/snapshots/churn-tests.v3.rb

```
def test_normal_subsystem_line_format
  assert_equal('         audit ********* (45)',
               subsystem_line("audit", 45))
end
```

The subsystem name is right-justified in a field fourteen characters wide, followed by a space, followed by nine asterisks and the count of 45.

What does this test tell us about the method we have to write? It will look something like this:

```
def subsystem_line(subsystem_name, change_count)
  # code here...
end
```

subsystem_name's string can be justified with Ruby's rjust method. That works like this:

```
irb(main):002:0> 'audit'.rjust(14)
=> "         audit"
```

Next the output has the row of nine asterisks. I'll put off figuring out how to make that string by assuming there's a method called asterisks_for that converts an integer into the right number of asterisks (nine, the number of changes divided by five). It'll be used like this: