Extracted from:

# Scripted GUI Testing with Ruby

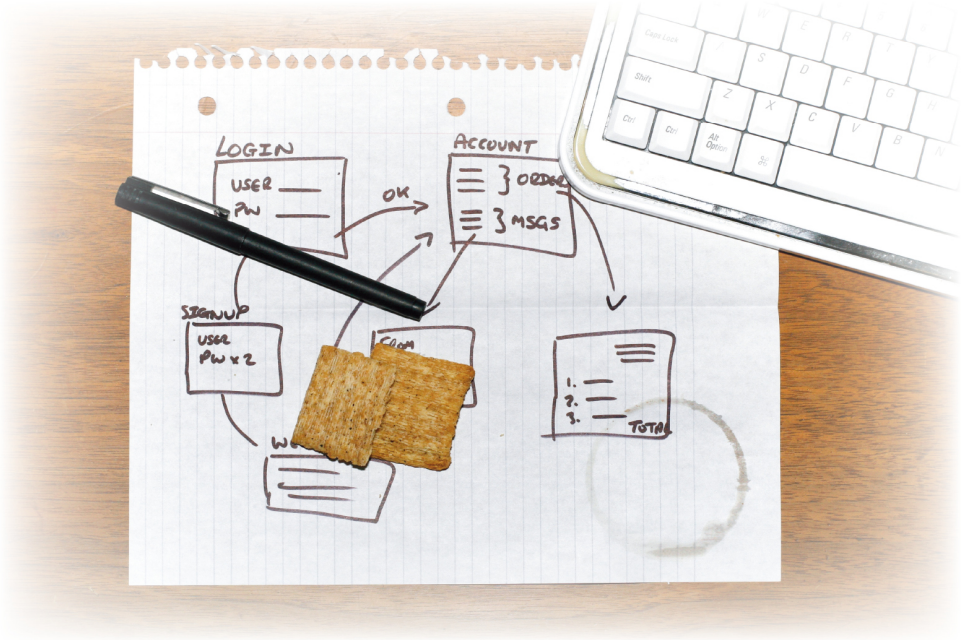


The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

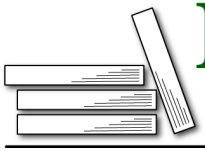
# Scripted GUI Testing with Ruby




*Ian Dees*

*Edited by Jacquelyn Carter*

The Facets of Ruby Series



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*.



Printed in the United States of America.
ISBN-13: 978-1-9343561-8-0
Printed on acid-free paper.
Book version: P1.1—January 2012

## 3.1 RSpec: The Language of Lucid Tests

Let's talk for a minute about the art of writing good test scripts. If we want our test code to be clear, it should be written in the application's problem domain—that is, using the same concepts that end users see when they use the software. In the case of LockNote, we should write scripts that deal in documents and passwords, not menu IDs and edit controls.

We also want to keep our test script from becoming one long, tangled, interdependent mess. So, we'll start with small, self-contained tests. Once we have confidence in our building blocks, we can assemble them into more meaningful tests.

During this process, it's helpful to think of these little units of test code as *examples* of correct behavior. I really mean it when I say we're going to start small. Our first examples will fit on a cocktail napkin.

### The Napkin

Imagine that you're sitting down for coffee with your software designers, chatting about how the program is going to work. Someone grabs a napkin, everyone huddles around talking and sketching excitedly, and you end up with something like Figure 3, *The ultimate requirements capture tool*, on page 6.

That kind of simplicity is just for sketches, right? Surely we have to abandon such hand-wavy descriptions when we actually start implementing our tests.

But what if we *could* write our test code the same way we wrote those notes on the napkin?

```
describe the main window
  it launches with a welcome message
  it exits without a prompt if nothing has changed
  it prompts before exiting if the document has changed
```

With just a handful of little examples like these, we could write about facets of our application's behavior in a specialized test description language. The language is easy to write and clear to read. There's just one problem: how do we get from paper to practice?

#### What Will This Buy Me?

What kinds of bugs will tests catch at this level of detail? Bad requirements, for one. When you fill in the bodies of those examples, your team will be forced to consider all kinds of usability edge cases as you describe how the app is really going to work.

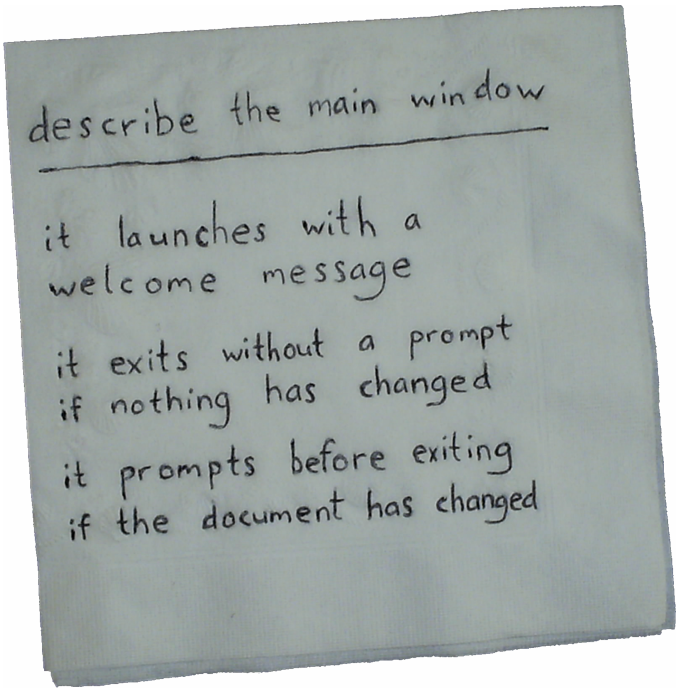



**Figure 3—The ultimate requirements capture tool**

You don't *need* a test script to do that. A sharp eye and empathy for your customer will help unearth the same kinds of issues.

But if you do choose to express your ideas as running code, you can press it into service later in the project as an automated "smoke test" that runs every time a developer checks in code.

### Introducing RSpec

The notation we've been using on this napkin is as real as Ruby. It's called RSpec.[1] It's implemented as a Ruby library, but you can also think of it as a language of its own—a *test description language* that just happens to be built on Ruby's strong metaprogramming foundation.[2]

The philosophy behind RSpec is that a good test should do more than exercise the code; it should also communicate its intentions clearly. RSpec provides two motifs for helping us write clear tests:

1. http://rspec.info
2. *Metaprogramming* is simply "programs writing programs." It's the technique that makes Ruby such a great platform for coders to build their own languages.

- The describe/it notation provides an overall structure for your test script.
- The should verb is how you write the individual pass/fail tests.

### describe/it

A few paragraphs ago, we saw that a good test script is more like a series of *examples* of correct behavior than an exhaustive specification. RSpec encourages this view of testing. Each example in RSpec is expressed as a sentence beginning with it, as in "it self-destructs when I hit the red button." We gather each group of related examples that describe one feature in, fittingly enough, a describe block.

It takes only a few keystrokes to transform our cocktail napkin into a set of RSpec examples:

```
describe 'The main window' do
  it 'launches with a welcome message'
  it 'exits without a prompt if nothing has changed'
  it 'prompts before exiting if the document has changed'
end
```

The code looks almost like it depends on some kind of fancy English language processing, but really it's just Ruby. describe() and it() are plain ol' Ruby functions supplied by the RSpec library.

We'll eventually fill in each of those it descriptions with specific tests, with help from RSpec's should idiom.

### should

In some testing toolkits, you're expected to use a system of "assertions" to write your pass/fail tests, something like this:

```
ASSERT_EQUAL(windowTitle, "My Program");
```

RSpec is a little different. Rather than asking you to make your style of writing more like programming, it bends the programming language to look more like natural writing. The previous example would look like this in RSpec:

```
window_title.should == 'My Program'
```

"Window title should equal 'My Program.'" You could practically read this code aloud. You could even show it to someone who's never seen Ruby before, and they'd probably understand what it does.

With RSpec, the should() and should_not() methods are available to every object in Ruby.[3] All of the following are valid tests in RSpec:

3. Thanks to Ruby's "open classes," whose definitions can be modified on the fly. This flexibility is what makes RSpec possible.

```
(2 + 2).should == 4
1.should be < 2
['this', 'list'].should_not be_empty
{:color => 'red'}.should have_key(:color)
```

Any test written with should() will raise an exception (and show up in the test report as a failed test) if its condition turns out to be false. Similarly, its companion method, should_not(), fails on true conditions.

Take a look at those last two tests. be_empty tells RSpec to call the empty?() method of the array. have_key calls the hash table's has_key?() method. This technique works for *any* method, not just empty?(). In general, be_xyz calls xyz?(), and have_xyz calls has_xyz?().

### Trying It

Let's grab the RSpec library and take it for a test-drive:

```
C:\> gem install rspec --version "~> 2.7"
```

Now our cocktail napkin translation is more than just a nicely formatted description of behavior. It's running code—try it! Save the code snippet (from *describe/it*, on page 7) as note_spec.rb, and run it with the spec executable, like this:

```
C:\> rspec --format=doc note_spec.rb

The main window
  launches with a welcome message (PENDING: Not Yet Implemented)
  exits without a prompt if nothing has changed (PENDING: Not Yet Implemented)
  prompts before exiting if the document has changed (PENDING: Not Yet Implemented)

Finished in 0.017212 seconds

3 examples, 0 failures, 3 pending
...
```

RSpec has noticed that our tests haven't been implemented yet. But we've definitely made progress. Three empty tests are better than no tests at all. Now, let's fill in those details.

### Putting It to Work

So far, our test script is merely an outline of what we will be doing. It describes which parts of the program we're testing, but it doesn't contain any pass/fail tests yet. Let's change that.

Remember our cautionary tale from the beginning of the chapter? We want to write our tests in the vocabulary of LockNote or JunqueNote and leave the platform-specific calls for a different part of the code. So, we're going to imagine that someone has lovingly provided a note-taking API just for us and code to that API. (Guess who's going to "lovingly provide" this API? Heaven helps those who help themselves....)

Replace the first it clause in your script with the following:

Download with_rspec/note_spec.rb

```
it 'launches with a welcome message' do
  note = Note.new
  note.text.should include('Welcome')
  note.exit!
end
```

The code at ① will create a new window (by launching the application). We'll add the implementation in a few minutes, using the automation techniques from the previous chapter.

At ②, we add our first actual pass/fail test. We want to make sure the word "Welcome" appears somewhere in the editable portion of the main window.

Finally, we shut down the app at ③. We'll follow the Ruby tradition of giving "dangerous" methods like exit!() an exclamation point. We want whoever is reading this code to know that the exiting program will discard the active document and steamroller over any save prompts along the way.

Now, when we run our script, we see the following:

```
1) The main window launches with a welcome message
   Failure/Error: note = Note.new
   NameError:
     uninitialized constant Note
   # ./note_spec.rb:4:in `(root)'
```

No surprise there. We've started tossing around this new term in our code, Note, without telling Ruby what it is. It's time to teach Ruby all about our note taking.

## 3.2 Building a Library

Up to this point, we've been working downward from our high-level test concepts to the specifics of LockNote and JunqueNote. Now it's time to build upward from the Windows and Java API calls we learned in Chapter 2, *An Early Success*, on page ?. We're going to put that low-level code together into a coherent library usable from our tests.

We want to do for our GUI tests what RSpec's creators did for testing in general: provide a way to express concepts clearly. RSpec will be our "gold standard" of beauty: we're going to shoot for a note-taking API clean enough to be at home inside an RSpec test.

### A Touch of Class

The code that we need to implement a clean API is already there in our two ..._basics.rb files; it just needs to be touched up a bit and organized into a Ruby class. We'll start with an empty class called Note in a new file named after the app we're testing (locknote.rb or junquenote.rb):

```
class Note
end
```

Later, we'll add each chunk of platform-specific calls as we find a good home for it.

To tell RSpec which program we're testing, we pass the name of the app with the -r option. So on Windows, we have this:

```
C:\> rspec -rlocknote -fd note_spec.rb
```

And for the cross-platform version, we have this:

```
$ jruby -S rspec -rjunquenote -fd note_spec.rb
```

What are the results when we try it?

```
1) The main window launches with a welcome message
   Failure/Error: note.text.should include('Welcome')
   NoMethodError:
     undefined method `text' for #<Note:0x34c57971>
   # ./note_spec_empty.rb:8:in `(root)'
```

As we expected, RSpec was able to create a Note object, but it couldn't do anything more. We haven't yet taught it to get the current document's text. In fact, we haven't even taught it to launch the application yet. Let's do so now.

### Starting Up

Reorganizing the code into a class will be pretty much the same whether you're playing the Windows or JRuby version of our home game.

Creating a new Note object should cause the app to launch. So, we'll move our window creation code from the previous chapter into Note's initialize() method:

```
«platform definitions»
class Note
  def initialize
    «code up through the first 'puts'»
  end
  «more to come...»
end
```

I won't show all the code here, because it's nearly an exact repeat of what you wrote in the previous chapter. You just put all your require lines (and Jemmy imports, for you JRuby readers) into the "platform definitions" section at the top and paste everything else up to the first puts into the body of initialize().

We'll use the main_window variable in some of the other methods we're defining, so we need to "promote" it to an attribute of the Note class. Replace main_window with @main_window everywhere you see it.

Now that we've taught our Note class how to launch the app, let's move on to text entry.

### Typing Into the Window

You've already written the code to simulate typing. It just needs to be made a bit more general. Grab the handful of lines that deal with keyboard input —look for "this is some text"—and paste them into a new type_in() method inside the Note class:

```
def type_in(message)
  «typing code here»
end
```

Of course, you'll probably want to replace the *"this is some text"* string literal with the message parameter that our top-level test script passes in. That takes care of writing text—how about reading it back?

### Getting Text Back from the Window

Up until now, we've been driving the GUI from our script, but we haven't retrieved any data from it yet. To change that state of affairs, we'll need one more platform-specific technique. It's an easy one, though, so I'm going to present the Windows and JRuby variants back-to-back.

#### Windows: The WM_GETTEXT Message

First, we want to drill down into LockNote's user interface and find the editable area that contains the document's text. This text area is a *child window* of the main window. To grab hold of it, we'll use FindWindowEx(). It's like the

FindWindow() function we used before, but with a couple of extra parameters —including the parent window option we need.

Once we've found the edit control, we'll send it the WM_GETTEXT message to find out what's inside it. You've seen the PostMessage() call for sending a message to a window. Its cousin SendMessage() is similar but is guaranteed to wait until the window actually *responds* to our message.

The meanings of SendMessage()'s parameters are different for every Windows message. For WM_GETTEXT, the last two parameters are the maximum size string we can accept and a pointer to the string where we want Windows to put the text we're asking for.

Here's what these two new API calls look like in use. Add the following code inside your LockNote class:

Download with_rspec/locknote.rb

```
def text
  find_window_ex = user32 'FindWindowEx', ['L', 'L', 'P', 'P'], 'L'

  send_message = user32 'SendMessage', ['L', 'L', 'L', 'P'], 'L'

  edit = find_window_ex.call @main_window, 0, 'ATL:00434310', nil

① buffer = '\0' * 2048
  length = send_message.call edit, WM_GETTEXT, buffer.length, buffer

  return length == 0 ? '' : buffer[0..length - 1]
end
```

As another concession to the manual memory management of the Windows world, we have to presize our buffer at ①, just like we did with get_window_rect() in the previous chapter.

### JRuby: The text Property

The JRuby approach to getting text is similar to the Windows one: we look for the editable text area (which belongs to the main window) and quiz it about its contents. Jemmy's JTextAreaOperator provides the text property for this purpose:

Download with_rspec/junquenote.rb

```
def text
① edit = JTextAreaOperator.new @main_window
  edit.text
end
```

The code at ① should look familiar; the type_in() method you wrote in the previous section contains one just like it. This is a sign that our code needs some cleanup, which we'll get to in the next chapter.

### Joe asks:
## What's the Significance of the Window Class?

In the previous chapter, we mentioned that a *window class* identifies whether a given window is a button, edit control, dialog box, or whatnot.

The basic controls that come with Windows have names like `edit` or `button`. This window class's name, `ATL:00434310`, is a little more complicated—it's a customization from Microsoft's open source Windows Template Library, used by LockNote's developers to write the application.

### Closing the Window

OK, Windows and Swing readers should both be ready for one final step in this chapter. Paste the remainder of your code into this skeleton:

```
def exit!
①  begin
     # ...remainder of code...

②    @prompted = true
③  rescue
   end
end
```

Windows users, you'll have to add one extra line at ①: paste in the definition of `find_window()` again just before the `begin`. We'll remove the need for this repetition soon.

Our higher-level test code will need to know if the program prompted us to save our document. So, we're going to wait for a few seconds for a save prompt to appear. If we see a prompt, we remember this event in the `@prompted` attribute at ②. If not, we'll get a `TimeoutError` (or `NativeException` in JRuby).

An exception isn't necessarily a bad thing in this case. It could be that we're exiting the app without changing anything—no need for a save prompt then. We just catch the exception at ③, and `@prompted` stays `nil`.

So, how do we use `@prompted` in our test script? As we discussed earlier, any test that reads `should have_xyz` will call a function named `has_xyz?()` and check its return value for `true` or `false`/`nil`.

```
def has_prompted?
  @prompted
end
```

### Two More Tests

We now have all the tools required to fill in the other two examples:

Download with_rspec/note_spec.rb

```
it 'exits without a prompt if nothing has changed' do
  note = Note.new
  note.exit!
  note.should_not have_prompted
end

it 'prompts before exiting if the document has changed' do
  note = Note.new
  note.type_in "changed"
  note.exit!
  note.should have_prompted
end
```

There you have it: one cocktail napkin turned into a working test plan.