Extracted from:

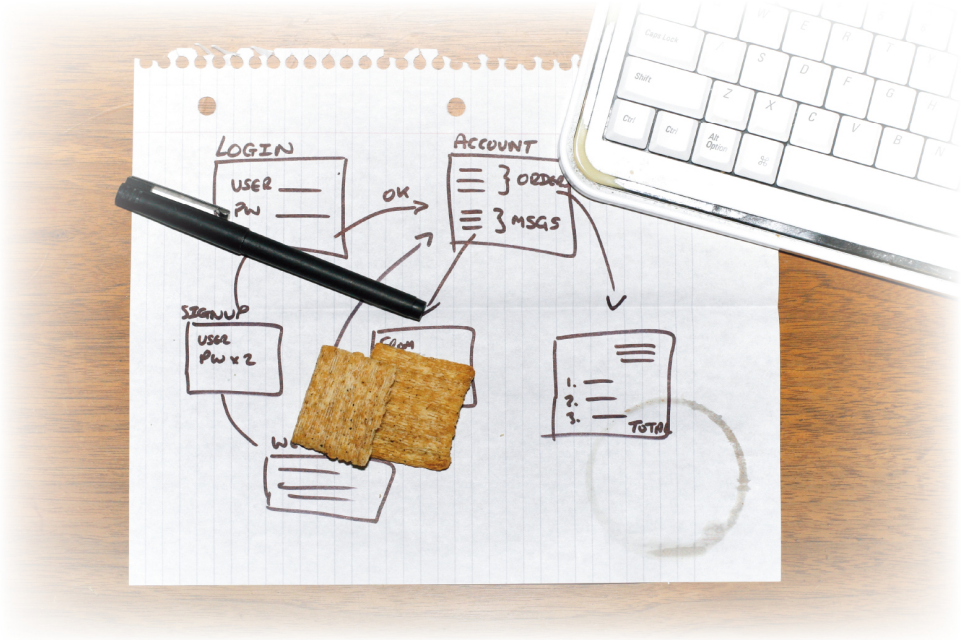
Scripted GUI Testing with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

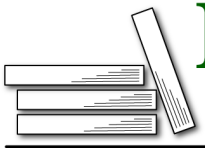
Scripted GUI Testing with Ruby



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

All right, no more Mr. Nice Tester. We've been coddling our application, gently running our test scripts the same way each time. That was fine for documenting our expectations of the app and getting our test framework up to speed.

But there could be dozens of bugs lurking in the program, waiting to be exposed if we'd only just do things with a little different order or timing. Let's try to trip up the app with a little randomness.

7.1 Keys, Menu, or Mouse?

In all our previous tests, we always exercise each feature the same way. For example, we always paste text by selecting the Paste item from the Edit menu. But there are at least two other ways to paste: pressing CTRL+V and using the right-click menu.

What if there were some weird interaction that caused problems with text manipulation, but only under certain unusual circumstances involving the keyboard? A lot of bugs are like that. If your test script always uses the menu, you'll never catch it.[1]

Keeping Things Interesting

Why not teach our script to act a bit differently each time? Let's start small with a single feature—Paste, for instance. Each time the test script calls paste(), we'll decide based on a random number whether to use the keyboard, menu bar, or context (right-click) menu:

Download guessing/locknote.rb

```
srand
① $seed ||= srand
srand $seed
puts "Using random seed #{$seed}"

class LockNote
  def paste
    case rand(3)
    when 0
      menu 'Edit', 'Paste'
      puts 'Pasting from the menu'
    when 1
      keystroke VK_CONTROL, 'V'.to_byte
      puts 'Pasting from a keyboard shortcut'
    when 2
② @main_window.click EditControl, :right
      type_in 'P'
```

1. Your manual tests *might* catch it, if you don't treat them like a rote "try all the menus" exercise.

```
      puts 'Pasting from a context menu'
    end
  end
end
```

At ②, we've added a new parameter to Window#click() to let us specify which mouse button we want to use. As in previous chapters, we identify the edit area by its window class. (EditControl is just a constant assigned to that hard-to-remember 'ATL:something' name.)

If we add code like this to some of our other functions, our test script will behave a little more like a real user: sometimes it'll use the mouse and sometimes the keyboard.

There is one potential downside to changing things up on each run like this. If your test happens to find a bug in the app, you might have trouble repeating the problem on the next run.

That's why it's important to record the value you use to seed Ruby's pseudo-random number generator, as we have at ①. Because srand() returns the *previous* seed value, it takes three calls to the function to get what we want.

Now, if we need to "play back" a particular sequence, we can put the seed we want in an external file, say, seed.rb…

Download guessing/seed.rb

```
$seed = 12345
```

and use it like this:

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

Adding that case/when code to every action that we want to randomize is going to get old really fast. Let's think about a way to avoid that kind of duplication.

Action!

What we're really doing in that case structure is defining all the different ways the Paste action could be carried out. Other actions in our software (Exit, Undo, Find) also have multiple ways for the user to perform them.

It makes sense to teach our test library the notion of defining an action, so we can do something like this:

Download guessing/locknote.rb

```
def_action :paste,
  :menu => ['Edit', 'Paste', :wait],
  :keyboard => [VK_CONTROL, 'V'.to_byte],
  :context => 'p'
```

The body of def_action() is just our Paste example from earlier, made more general:

Download guessing/locknote.rb

```
class LockNote
  @@default_way = :random

  def self.def_action(name, options, way = nil)
    define_method name do
      keys = options.keys.sort {|k| k.to_s}

①     way ||= @@default_way
      key = case way
        when nil;     keys.last
        when :random; keys[rand(keys.size)]
        else          way
      end

      action = options[key]

      case key
      when :menu
        menu *action
        puts "Performing #{name} from the menu bar"

      when :keyboard
        keystroke *action
        sleep 0.5
        puts "Performing #{name} from a keyboard shortcut"

      when :context
        @main_window.click LockNote::EditControl, :right
        sleep 0.5
        type_in action
        sleep 0.5
        puts "Performing #{name} from a context menu"

      else
        raise "Don't know how to use #{key}"
      end
    end
  end
end
```

Notice how, at ①, we've turned the preferred type of action into what amounts to a configuration setting. You could use :random for overnight stress testing, :keyboard to run a few tests on a mouse-free machine, or something else (e.g., :preferred) for times when you need predictability.

Equipped with this class-level method, we can define several common GUI actions:

Download guessing/locknote.rb

```
def_action :undo,
  :menu => ['Edit', 'Undo', :wait],
  :keyboard => [VK_CONTROL, 'Z'.to_byte]

def_action :cut,
  :menu => ['Edit', 'Cut', :wait],
  :keyboard => [VK_CONTROL, 'X'.to_byte],
  :context => 't'

def_action :copy,
  :menu => ['Edit', 'Copy', :wait],
  :keyboard => [VK_CONTROL, 'C'.to_byte],
  :context => 'c'

def_action :delete,
  :keyboard => [VK_BACK],
  :context => 'd'

def_action :select_all,
  :menu => ['Edit', 'Select All', :wait],
  :keyboard => [VK_CONTROL, 'A'.to_byte],
  :context => 'a'
```

Our script has just been given a bit more bug-finding potency, but it's also more expressive now. It almost reads like documentation: "The Undo action can be triggered from the Edit > Undo menu or the CTRL+Z keystroke."

Decluttering

Although our test library has become more readable, our test report has become a mess:

```
Using random seed 12345

The editor
Performing select_all from the menu bar
Performing delete from a keyboard shortcut
Performing select_all from a keyboard shortcut
Performing cut from a keyboard shortcut
Performing select_all from a context menu
Performing delete from a keyboard shortcut
Performing select_all from the menu bar
Performing paste from the menu bar
- supports cutting and pasting text

Finished in 13.809 seconds
```

```
1 example, 0 failures
```

Let's use Ruby's logging library so we can separate the test report from the extra info:

Download guessing/locknote.rb

```
require 'logger'

① class SimpleFormatter < Logger::Formatter
    def call(severity, time, progname, msg)
      msg2str(msg) + "\n"
    end
  end

  $logger = Logger.new STDERR
② $logger.formatter = SimpleFormatter.new
```

Logger's default format is pretty verbose: it includes a time stamp, a logging level, and so on. At ① and ②, we've trimmed it to just a simple description.

Now, if we replace all our calls to puts(), like this one…

```
puts "Performing #{name} from the menu bar"
```

with calls to $logger.info(), like this…

```
$logger.info "Performing #{name} from the menu bar"
```

then all the extra information not part of the test report will be directed to wherever the Logger is pointed, in this case STDERR. On Windows, you can redirect standard error to a file with the 2> operator:

```
C:\> rspec -rlocknote -fd note_spec.rb 2>actions.txt
```

If one of our tests suddenly fails in the face of randomness, we now have the random number seed to re-create it and a detailed record of GUI actions to help us diagnose it.