Cucumber meets iPhone

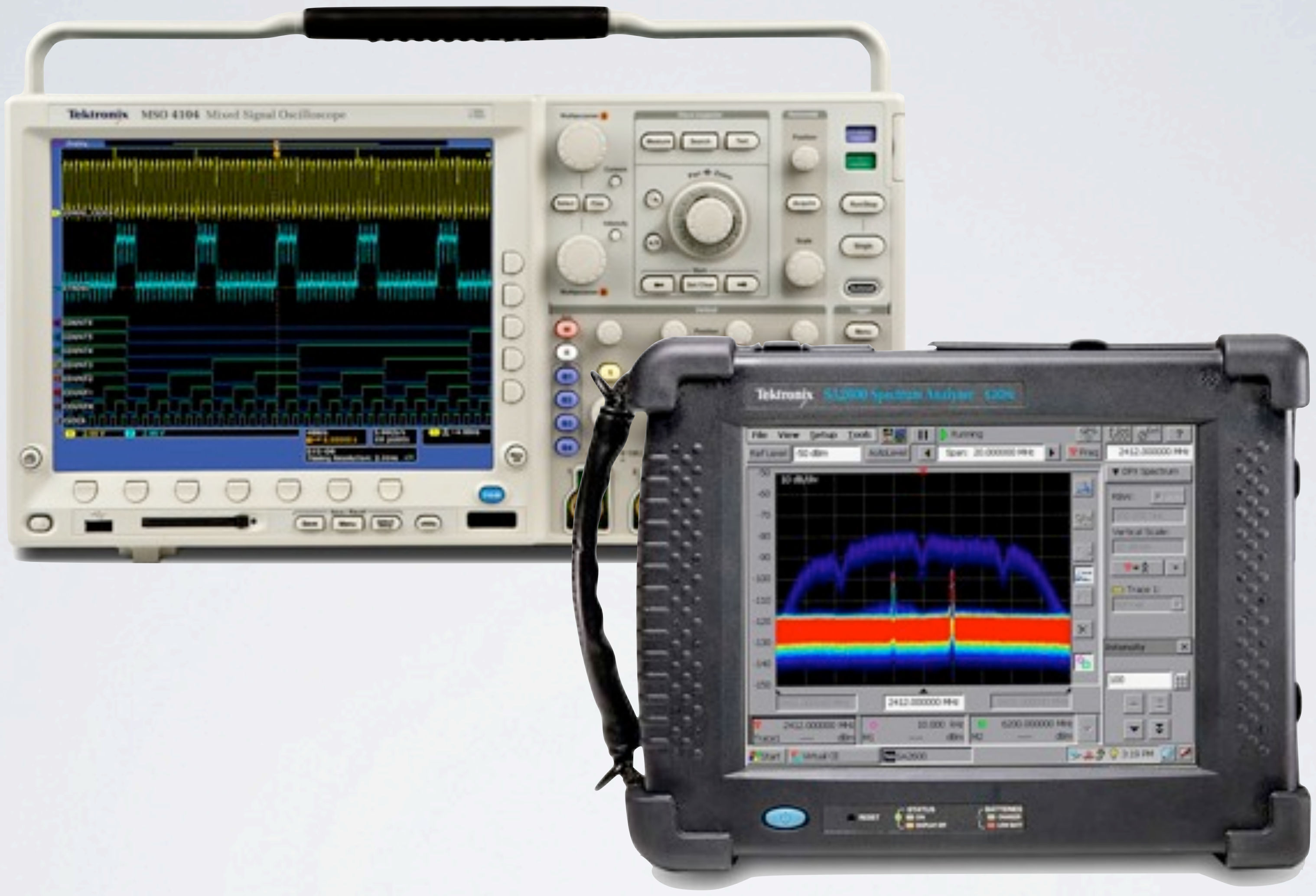
OSCON 2009

Ian Dees
http://www.ian.dees.name

Hi, everyone. Welcome to OSCON. This talk is about testing the GUIs of iPhone apps using the Ruby language and the Cucumber test framework.

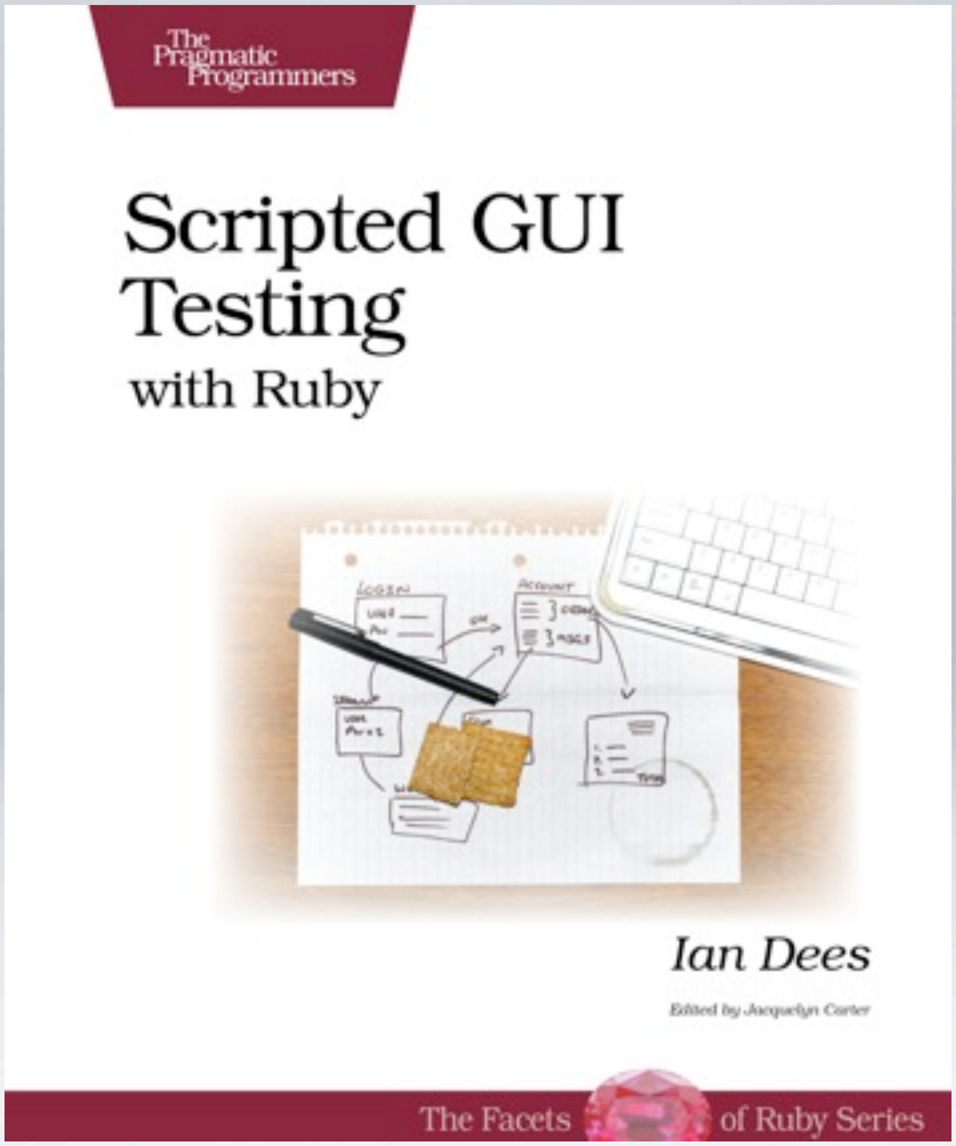
#osconcuke

First, a bit of business. Per Giles Bowkett, here’s a Twitter hashtag for this talk. Throughout the presentation, audience members can tweet their questions here. This will hopefully keep people from being put on the spot, and will also provide a place to fill in any answers we don’t have time for today.



Tektronix

Now, a bit about what I do—because you're no doubt deeply caring and curious people, and because it will shed light on what we're doing here today. I write software for Tektronix, makers of test equipment (though this talk isn't about my day job). The interfaces we build involve things like touch screens and physical knobs. Each style of interface demands new ways of testing.


The Pragmatic Programmers

Scripted GUI Testing with Ruby

Ian Dees

Edited by Jacquelyn Carter

The Facets of Ruby Series

I’ve written a book that tells that kind of story—adapting tests to the quirks and realities of a platform—in code.

When a new platform comes out, be it a phone or a runtime or an OS, I get curious: how do they test that? And that's what led me to think about the kinds of things I'm talking about today. So, on to our topic: testing the iPhone with Cucumber.

CUCUMBER

http://cukes.info

So, Cucumber.

LET'S GET **U** INTO BEHAVIOUR-DRIVEN DEVELOPMENT

Cucumber is a Behavio(u)r-Driven Development framework built on Ruby. You write your top-level tests in a language suited for that task (English). And you write your glue code in a language suited for that task (Ruby).

FLAVOUR OF THE MONTH?

Flavor of the month? (Buzzword) bingo! BDD is indeed a flavor, a seasoning, a heuristic which you use insofar as it helps you make sense of your projects.

EXECUTABLE EXAMPLES

For the purposes of this talk, think of BDD as an appreciation of executable examples. Early conversations about software projects tend to take the form of examples and use cases; why not express them as running code and have some smoke tests you can use later?

Ya gotta have a real project to throw your ideas against.
—Gus Tuberville (heavily paraphrased)

It’s all well and good to add a couple of token tests to a “Hello, world” project. But it’s much more instructive to actually try to solve some real problem.

QUESTION

"Oh crap, OSCON accepted my talk submission. *Now* what do I do?"

So I looked for a real-world problem to solve.

QUESTION

"Can BDD be used to write an iPhone app?"

Something that wasn't related specifically to this conference, or even to the general task of seeing how well BDD and iPhone development fit together.

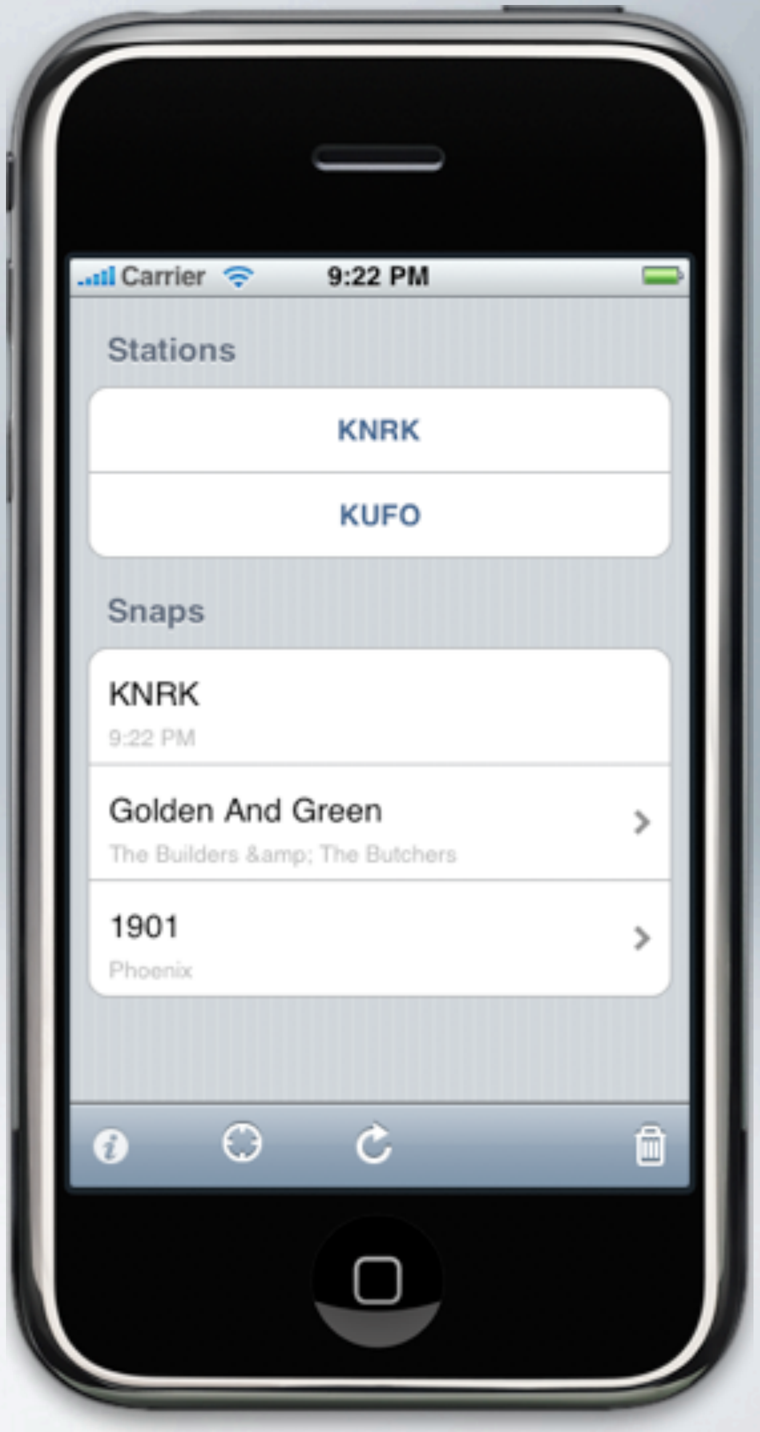
QUESTION

"What's on the radio?"

It had to be something I would have liked to do anyway, irrespective of conferences and technologies. Something to keep the imagination circuits grounded in reality.

JUST PLAYED

http://www.25y26z.com



So I undertook to write an iPhone app that would let me “bookmark” what radio station is playing any given moment and look the information up later. Yes, I’ve heard of pencil and paper, Shazam, and the Zune. There are tradeoffs that make this app better than those approaches in some ways, and worse in others.

HOW THIN IS YOUR GUI?

Presenter First Technique—Brian Marick
http://exampler.com/src/mwrc-start.zip

Do you need a full-stack GUI test? If your GUI is an ultra-thin, ultra-reliable layer on top of an API, many of your automated tests can bypass the GUI and talk directly to the API.

DRIVING A GUI

1. Push the button
2. See what happens

At some point, though, it’s nice to see the real thing running, top to bottom. And that’s the approach I took with the Just Played app.

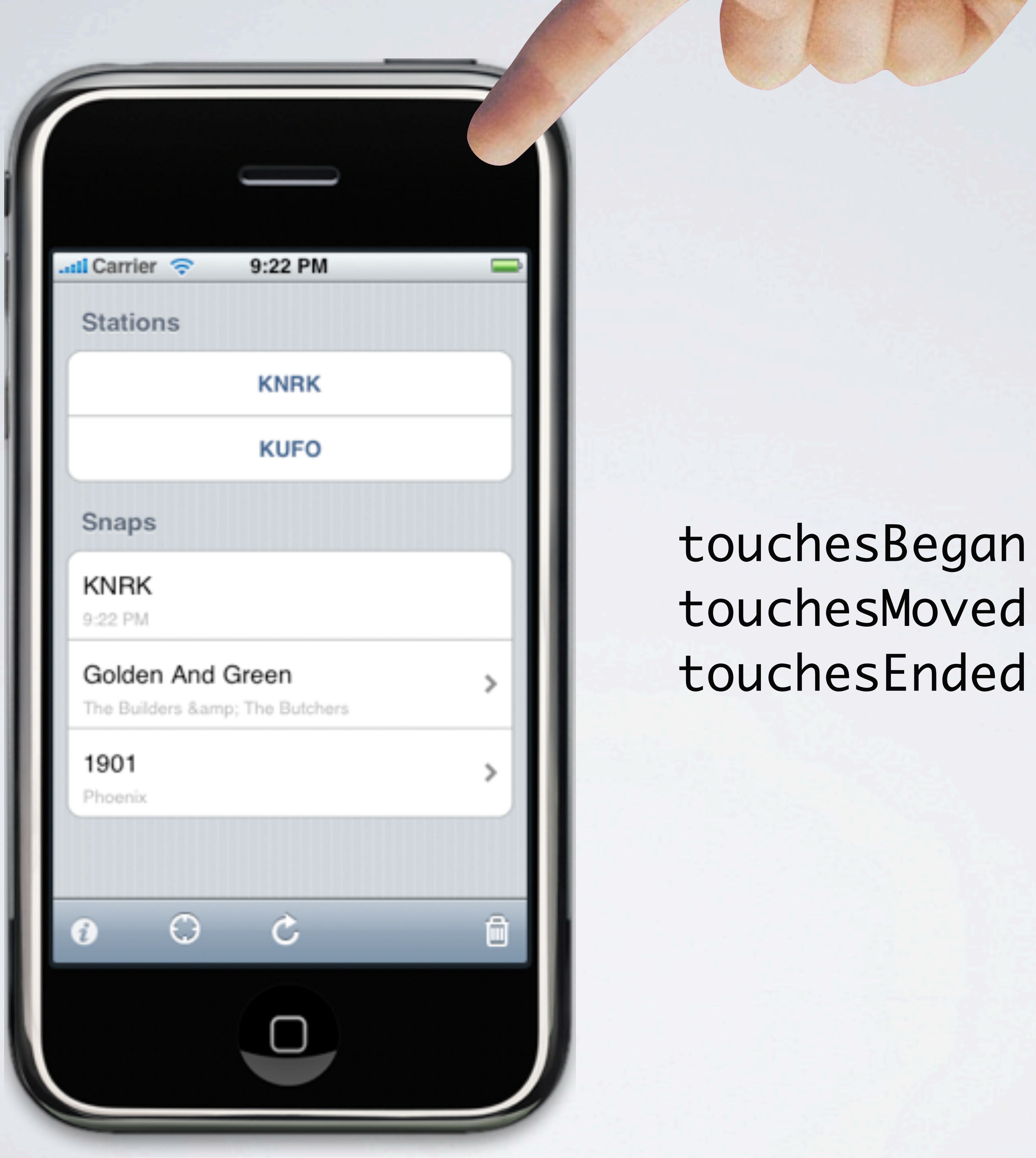
The two main phases of automating a GUI are simulating events and inspecting controls.

DRIVING THE PHONE

Matt Gallagher

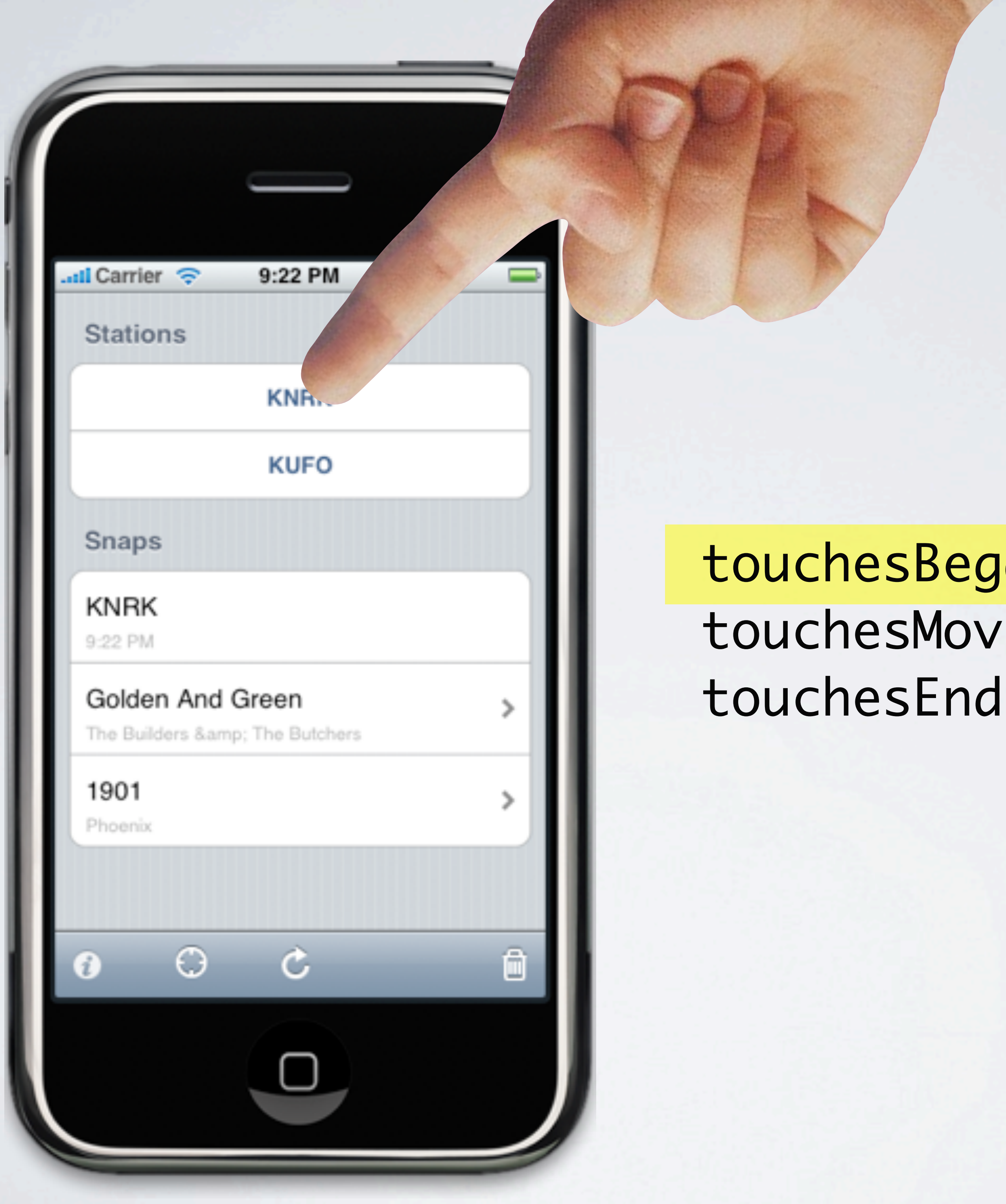
http://cocoawithlove.com/2008/11/automated-user-interface-testing-on.html

Matt Gallagher has done the groundbreaking work on both these fronts. In a series of articles on his Cocoa With Love blog, he discusses how to synthesize touch screen events and interrogate the GUI on an iPhone.



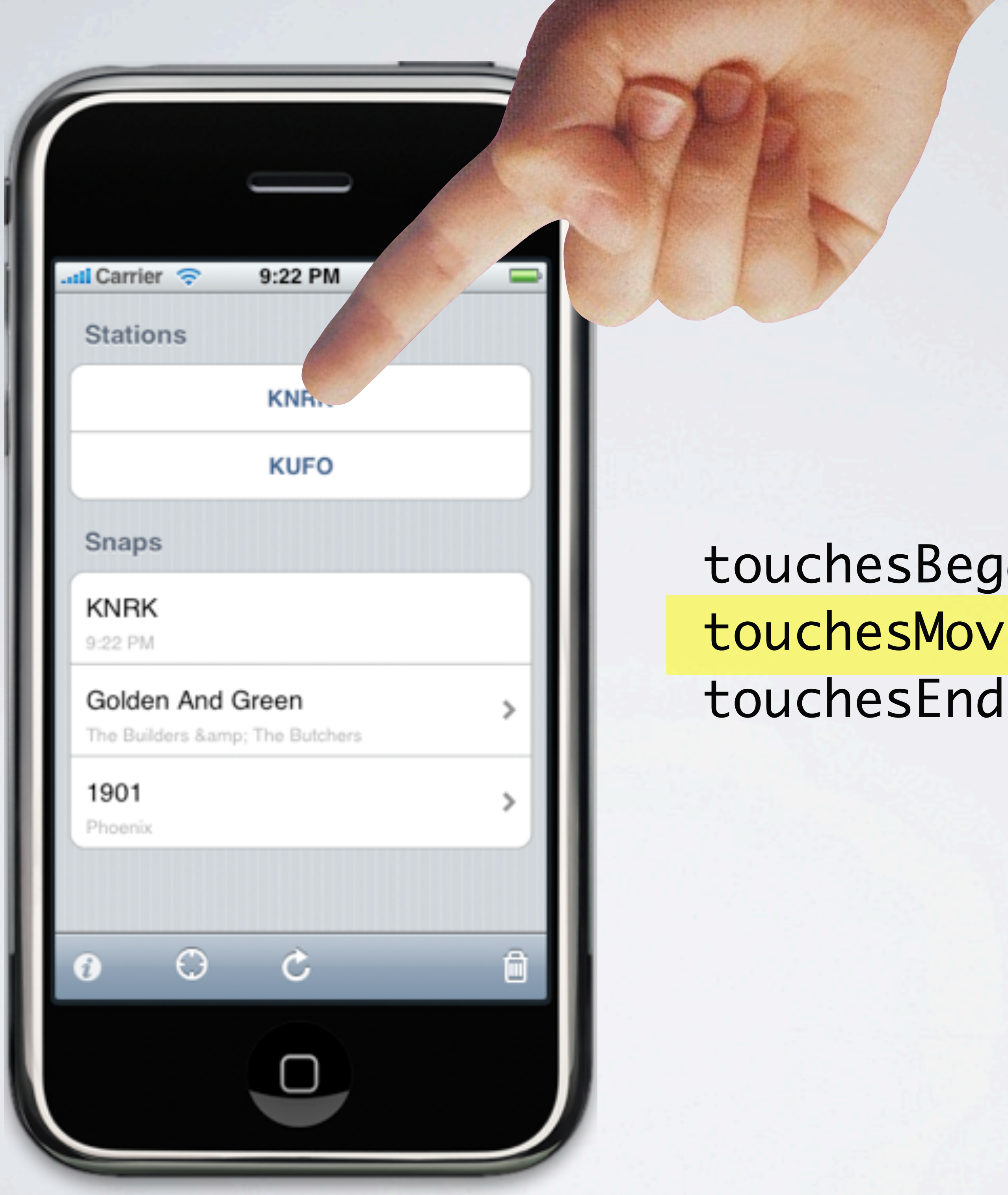
```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
```

Here’s how Matt’s technique works. When you tap a button on an iPhone, the Cocoa Touch runtime looks at the button’s implementation for these three methods and calls them in sequence...



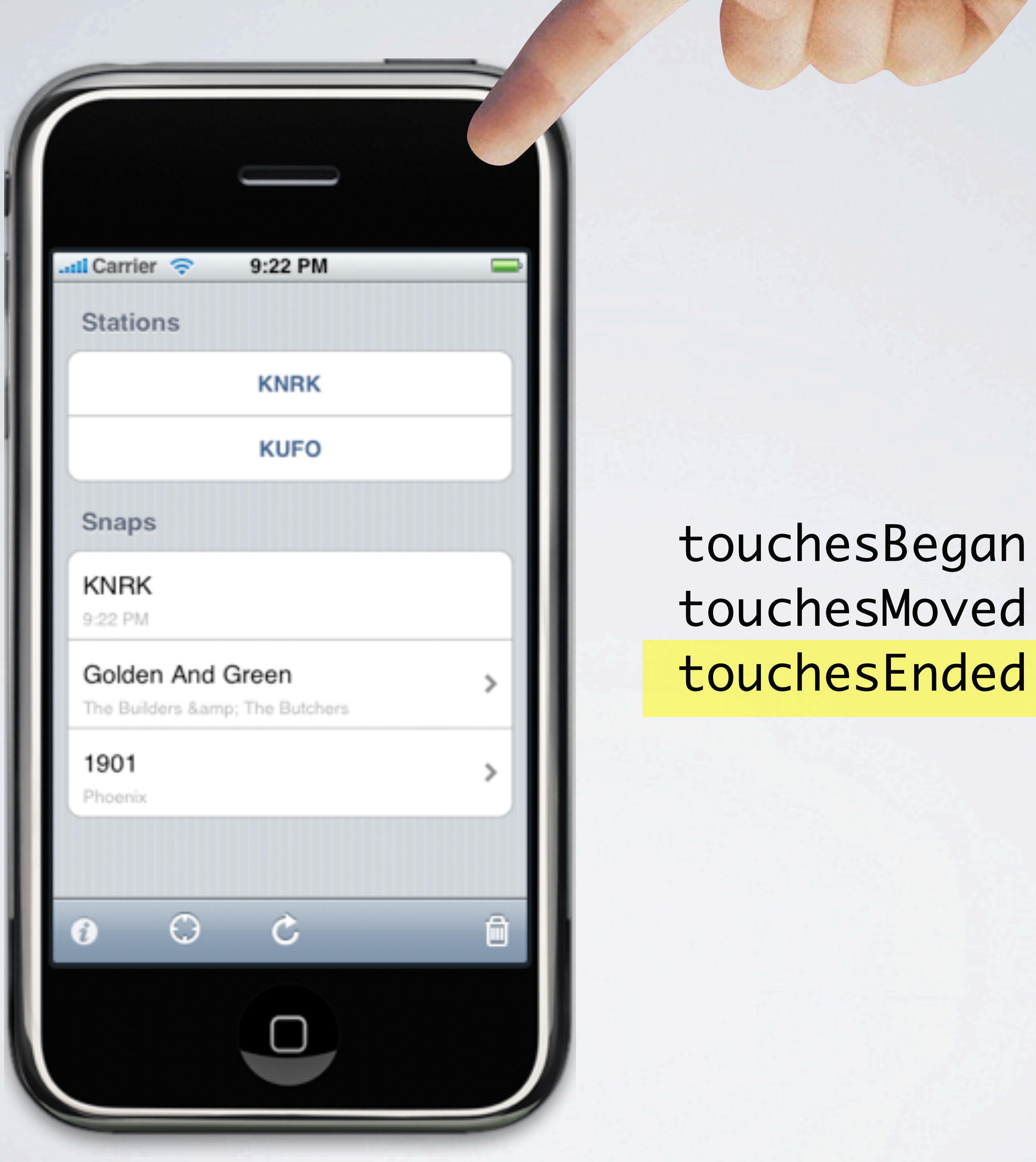
```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
```

...



```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
```

...



```
touchesBegan:withEvent:
touchesMoved:withEvent:
touchesEnded:withEvent:
```

...

WISH WE WERE HERE

```
UITouch *touch   = [[[UITouch alloc] init] autorelease];
NSSet   *touches = [NSSet setWithObject:touch];
UIEvent *event   = [[[UIEvent alloc] init] autorelease];

[view touchesBegan:touches withEvent:event];
[view touchesEnded:touches withEvent:event];
```

So all we have to do is create the UITouch and UIEvent objects that the functions expect to see, and all will be well, right? Except that Apple didn't provide initializers for these objects. And they have private fields that have to be set up just so, or your app will crash.

CATEGORIES

```
@interface UITouch (Synthesize)

- (id)initInView:(UIView *)view;
- (void)setPhase:(UITouchPhase)phase;
- (void)setLocationInWindow:(CGPoint)location;
- (void)moveLocationInWindow;

@end
```

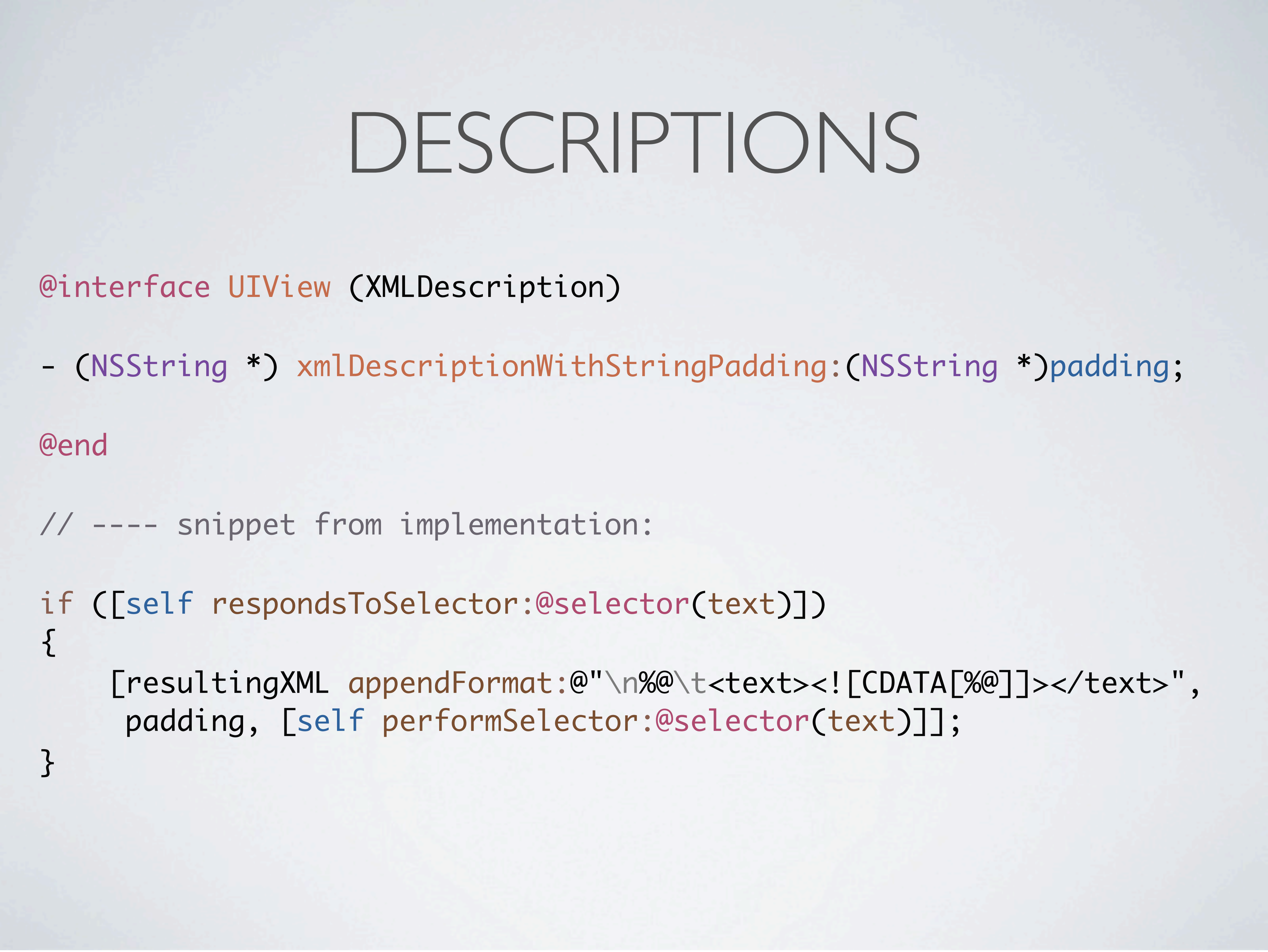
In Ruby, you'd write a monkey patch, modifying the vendor's class at runtime. Objective-C offers a slightly more civilized take on the concept, called a category. You can add methods to a third-party class, and spell out the scope in which they apply. Your new methods have access to the class's private fields; lucky for would-be GUI testers!

```
- (id)initInView:(UIView *)view
{
    // ... code omitted for brevity ...
    _tapCount = 1;
    _locationInWindow =
        CGPointMake(
            frameInWindow.origin.x + 0.5 * frameInWindow.size.width,
            frameInWindow.origin.y + 0.5 * frameInWindow.size.height);
    _previousLocationInWindow = _locationInWindow;

    UIView *target =
        [view.window hitTest:_locationInWindow withEvent:nil];

    _window = [view.window retain];
    _view = [target retain];
    _phase = UITouchPhaseBegan;
    _touchFlags._firstTouchForView = 1;
    _touchFlags._isTap = 1;
    _timestamp = [NSDate timeIntervalSinceReferenceDate];
    // ... code omitted for brevity ...
}
```

I gather this wasn't easy. It probably involved making an educated guess, reading the crash logs, trying to read Apple's minds, making some tweaks, and watching it crash all over again. Here are just some of the private fields whose purposes, or at least required values, Matt was able to divine.



DESCRIPTIONS

```
@interface UIView (XMLDescription)

- (NSString *) xmlDescriptionWithStringPadding:(NSString *)padding;

@end

// ---- snippet from implementation:

if ([self respondsToSelector:@selector(text)])
{
    [resultingXML appendFormat:@"\n%@\t<text><![CDATA[%@]]></text>",
     padding, [self performSelector:@selector(text)]];
}
```

The same technique—patching existing classes at runtime—works for implementing the other half of the GUI testing equation. Here’s a glimpse at how categories put a uniform face on runtime descriptions: location, type, caption, and other information for every control on the screen.

SCRIPT RUNNER

But how do you specify which button gets clicked, and when, and what the expected outcome is? The original approach provides a ScriptRunner object that reads in a list of commands from a text file.

BROMINE

(not the openqa one)
Felipe Barreto
http://bromine.googlecode.com

Matt’s original proof-of-concept embedded the test script inside a special build of an app, which ran “headless”—working tests, but no GUI. Felipe Barreto packaged up the code for running with a visible GUI, in the simulator or on the iPhone. The project is called Bromine (as a nod to the Selenium web testing tool), and is drop-in ready.

```
<plist version="1.0">
<array>
    <dict>
        <key>command</key>
        <string>simulateTouch</string>
        <key>viewXPath</key>
        <string>//UIToolbarButton[1]</string>
    </dict>
</array>
</plist>
```

The test script format for Bromine, like Matt’s project before it, is an XML-based property file. It’s simple to parse from Cocoa Touch, but it requires re-bundling your app every time you change your test script. Also, adding conditional tests requires custom Objective-C code.

```
curl --url http://localhost:50000 -d       \
"<plist version=\"1.0\">                   \
 <dict>                                    \
     <key>command</key>                    \
     <string>simulateTouch</string>        \
     <key>viewXPath</key>                  \
     <string>//UIToolbarButton[1]</string> \
 </dict>                                   \
 </plist>"
```

So just for fun, I found an open-source Objective-C HTTP server, dropped it into the project, and taught Bromine how to read its commands from the server instead of a file. Now you can keep your app running and drive it from curl.

```ruby
require 'tagz'

command = Tagz.tagz do
  plist_(:version => 1.0) do
    dict_ do
      key_    'command'
      string_ 'simulateTouch'
      key_    'viewXPath'
      string_ '//UIToolbarButton[1]</string>'
    end
  end
end

Net::HTTP.post_quick \
  'http://localhost:50000/', command
```

Or Ruby.

BROMINE + NET = BROMINET

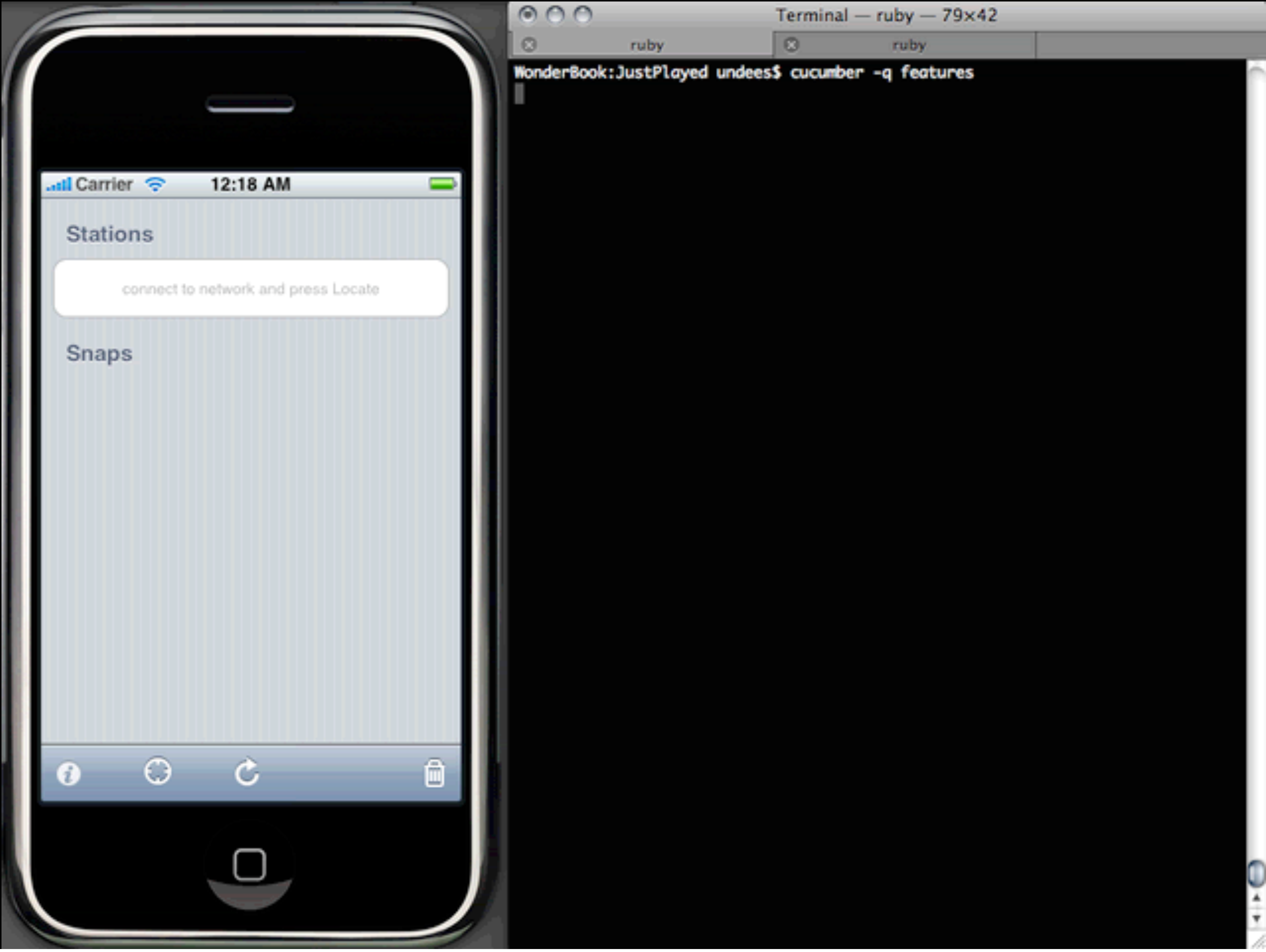
http://github.com/undees/brominet

I lightheartedly call the result “Brominet,” as a nod to the new-found network connectivity (and because it’s missing a bunch of functions the full Bromine library has).

COMPATIBILITY

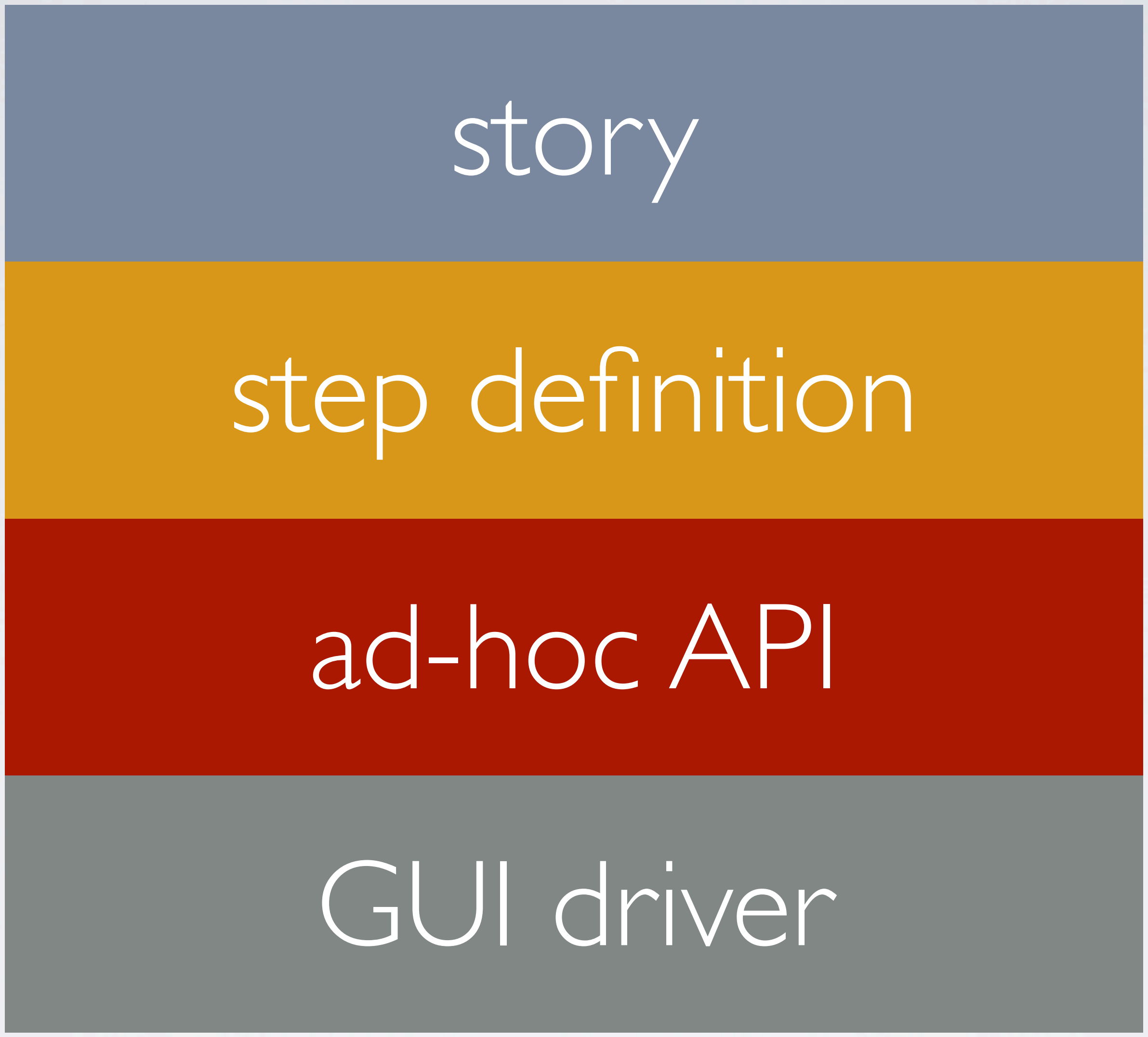
	2.2	3.0
Simulator	✓	×
iPhone	✓	×

One of the drawbacks of using categories to set private fields is that the vendor might change those fields out from under you. The Bromine project is working on getting the library updated for iPhone OS 3.0, and I’ll port any updates over to Brominet.



```
WonderBook:JustPlayed undees$ cucumber -q features
```

We’ve now got enough pieces to look at some of the test scripts for the Just Played app, top to bottom.



The top-level user stories are in Cucumber's test-oriented subset of English. The Ruby code to press specific buttons is effectively an ad-hoc, informally specified, bug-ridden API for the app. A layer of glue code—step definitions—sits between them. Down in the plumbing, we make angle brackets and post them via HTTP.

```
Feature: Stations
  In order to keep my day interesting
  As a radio listener with varied tastes
  I want to track songs from many stations

  Scenario: Deleting stations
    Given a list of radio stations "KNRK,KOPB"
    When I delete the first station
    Then I should see the stations "KOPB"
```

S

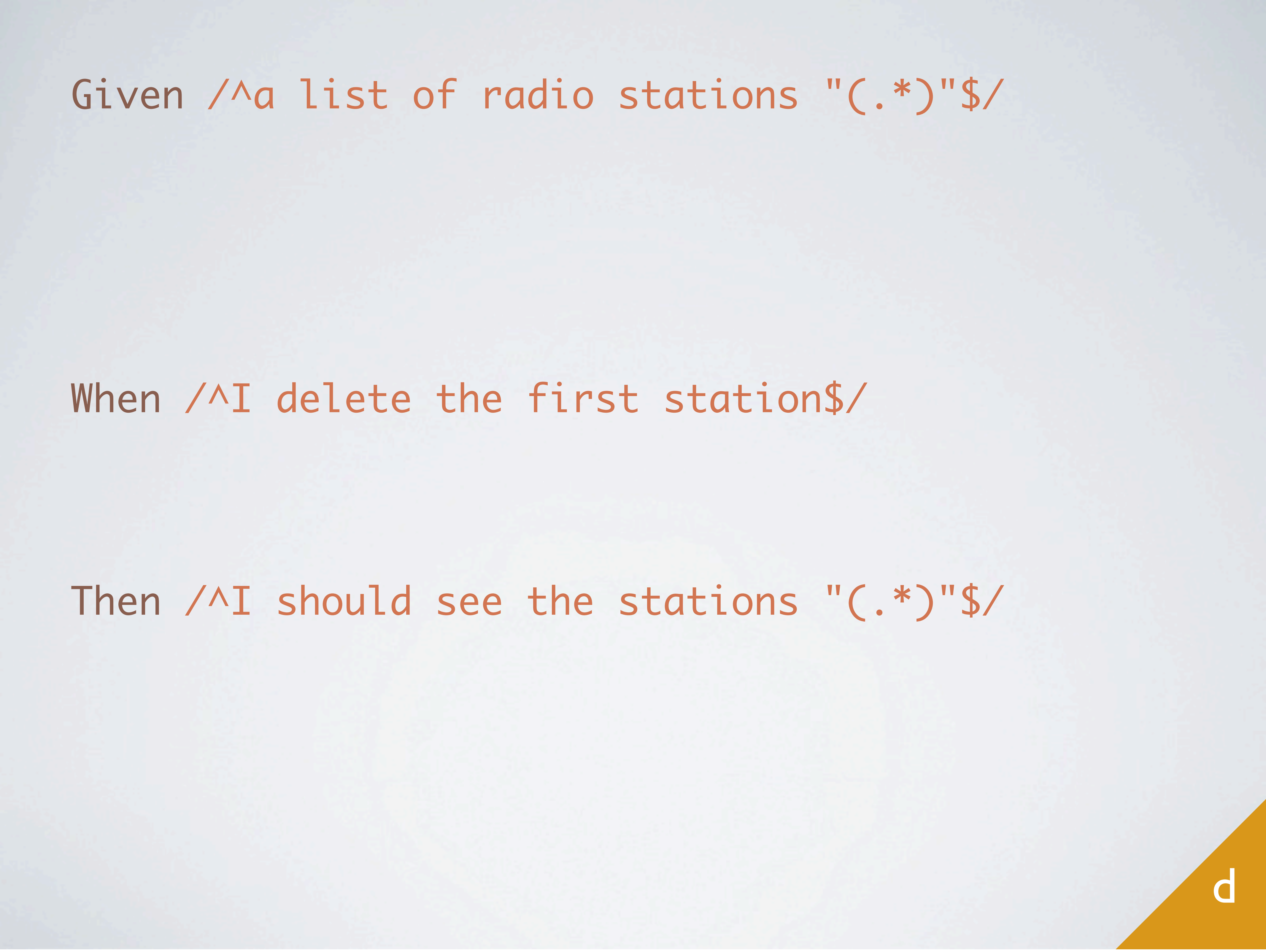
Here's a story from late in the app's development. As with all the features, I wrote the story first, and the feature later. The first few lines are just commentary, imploring the test to justify its own existence.

```
Given a list of radio stations "KNRK,KOPB"

When I delete the first station

Then I should see the stations "KOPB"
```

The real heart of the tests are the Given/When/Then lines.



```
Given /^a list of radio stations "(.*)"$/

When /^I delete the first station$/

Then /^I should see the stations "(.*)"$/
```

Cucumber searches among the Ruby step definitions you’ve provided to find the match for each step.

```
Given /^a list of radio stations "(.*)"$/ do



end

When /^I delete the first station$/ do

end

Then /^I should see the stations "(.*)"$/ do




end
```

d

When Cucumber hits a line of your test script, it runs the block of code attached to the matching step definition. We could put HTTP POST requests directly in these blocks.

```
Given /^a list of radio stations "(.*)"$/ do
  |stations|

  app.stations = stations.split(',')
end

When /^I delete the first station$/ do
  app.delete_station(0)
end

Then /^I should see the stations "(.*)"$/ do
  |text|

  stations = text.split ','
  app.stations.should == stations
end
```

d

But it’s often useful to wrap the specific button presses up in an abstraction. Rather than say, “press the button whose caption is ‘Delete,’” you say, “delete something,” and rely on your ad-hoc API to define how deleting happens in the GUI.

```
class JustPlayed
  def stations=(list)
    plist = Tagz.tagz do
      array_ do
        list.each {|station| string_ station}
      end
    end

    @gui.command 'setTestData', :raw,
      'stations', plist
    end
  end
end
```

a

And here are a few pieces of that API. First, for the Given step, we're cheating and pre-loading the GUI with canned data. (In the context of Rails apps, David Chelimsky calls this technique Direct Model Access.) Brominet sees the "setTestData" command, recognizes it as something not GUI-related, and hands it off to the app's delegate class.

```
class JustPlayed
  def delete_station(row)
    @gui.command 'scrollToRow',
      'viewXPath', '//UITableView',
      'rowIndex', row

    @gui.command 'simulateSwipe',
      'viewXPath', "//UITableViewCell[#{row + 1}]"

    @gui.command 'simulateTouch',
      'viewXPath', '//UIRemoveControlTextButton',
      'hitTest', 0

    sleep 1
  end
end
```

a

Once the GUI has the canned test data, we can push some buttons. Two things to note: 1) Bromine can scroll controls into view so they can be tapped, and: 2) since Bromine sees the view hierarchy as one big XML tree, we specify controls using XPath.

```ruby
class JustPlayed
  def stations
    xml = @gui.dump
    doc = REXML::Document.new xml

    xpath = '//UITableViewCell[tag="%s"]' % StationTag
    titles = REXML::XPath.match doc, xpath
    titles.map {|e| e.elements['text'].text}
  end
end
```

a

To get the result, we just serialize the entire GUI and grub through it in the comfort of the desktop. This process is slow, but allows Brominet to be built on a small set of primitives.

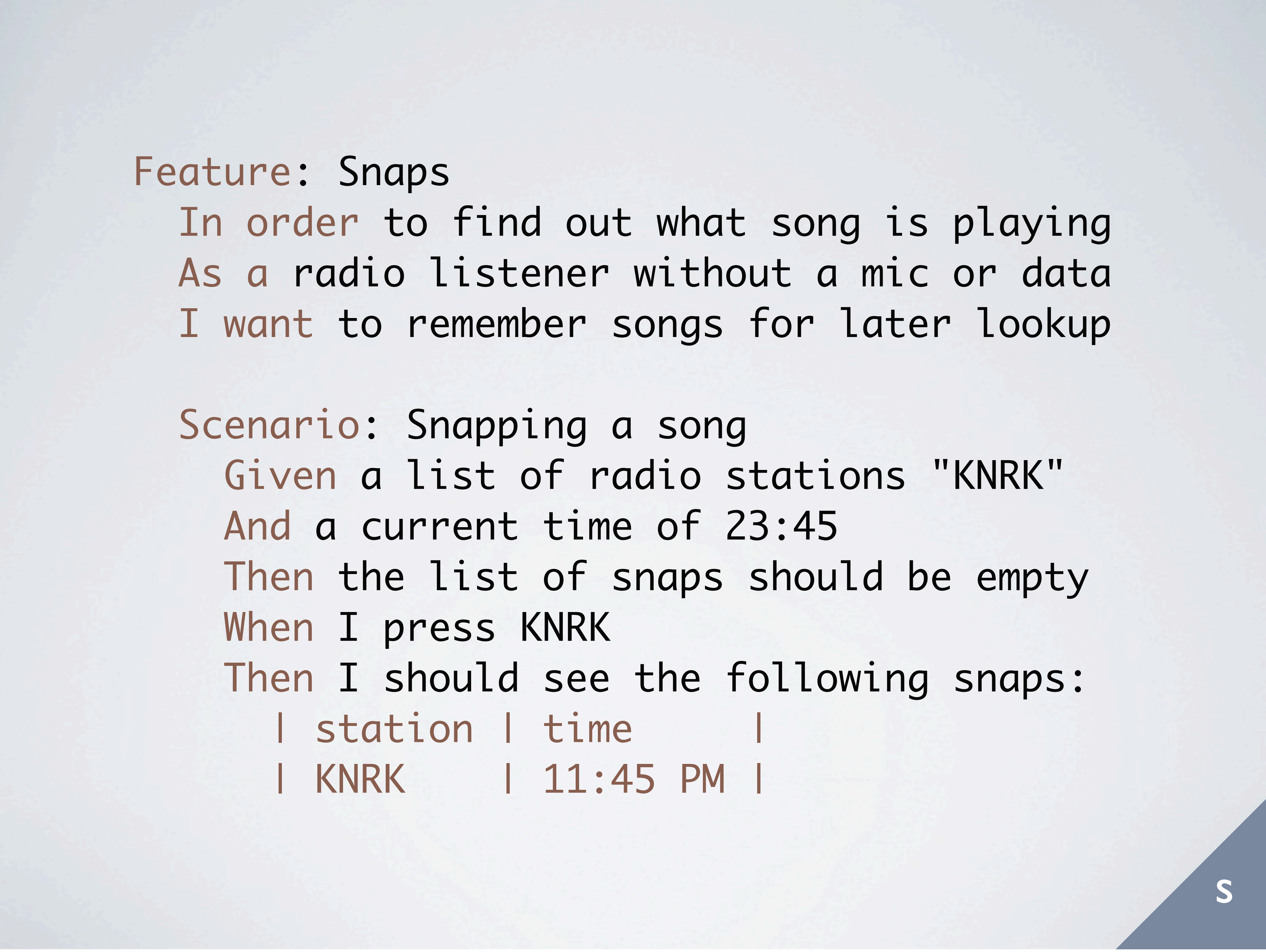
```
module Encumber
  class GUI
    def command(name, *params)
      command = Tagz.tagz do
        plist_(:version => 1.0) do
          dict_ do
            key_ 'command'
            string_ name
            params.each_cons(2) do |k, v|
              key_ k
              string_ v
            end
          end
        end
      end

      Net::HTTP.post_quick \
        "http://#{@host}:#{@port}/", command
    end
  end
end
```

All those library functions call into a common class whose responsibility is posting HTTP requests. It uses the Tagz library to build XML requests and Net::HTTP to send them. This part of the code should be reusable in any Brominet project.

FAKING THE TIME

The canned-data approach is about to rear its head again, as we're going to fake out parts of the physical world.



```
Feature: Snaps
  In order to find out what song is playing
  As a radio listener without a mic or data
  I want to remember songs for later lookup

  Scenario: Snapping a song
    Given a list of radio stations "KNRK"
    And a current time of 23:45
    Then the list of snaps should be empty
    When I press KNRK
    Then I should see the following snaps:
      | station | time     |
      | KNRK    | 11:45 PM |
```

This is the first story I committed to the repository, even before creating a new Xcode project. (The story has, of course, been edited several times since then.)

See the line about the current time of 23:45?

```
Given /^a current time of (.*)$/ do
  |time|

  app.time = time
end
```

d

Without having control over the time, we’d have to resort to creating a bookmark and then checking the time right after, with some sort of fuzzy matching.

```
class JustPlayed
  def time=(time)
    @gui.command 'setTestData',
     'testTime', time
  end
end
```

The app has to support this testing hook internally and worry about whether it's supposed to use the real time or the fake one. This juggling introduces complexity. But you're seeing the worst case. The preloading of radio stations you saw earlier met with a much happier fate: the support code was useful for implementing save/restore.

TABLES

One of Cucumber’s ballyhooed features is its support for tabular data.

```
Feature: Lookup
  In order to find out what songs I heard earlier
  As a radio listener with network access
  I want to look up the songs I bookmarked

Scenario: Partial success
    Given a list of radio stations "KNRK"
    And a test server
    And the following snaps:
      | station | time       |
      | KNRK    | 2:00 PM    |
      | KNRK    | 12:00 PM |
    When I look up my snaps
    Then I should see the following snaps:
      | title                 | subtitle         |
      | KNRK                  | 2:00 PM          |
      | Been Caught Stealing | Jane's Addiction |
```

You can draw ASCII-art tables right in your stories.

```
Given /^the following snaps:$/ do
  |snaps_table|

  snaps_table.hashes
    #=> [{'station' => 'knrk', 'time' => '2:00 PM'} ...]

  # do fancier things, like manipulating columns
  app.snaps = snaps_from_table(snaps_table, :with_timestamp)
end
```

d

If a test step comes with its own table, Cucumber passes it into the definition block as an object you can poke at. Most of the time, you'll just want to grab its "hashes" array, which gives you one hash per row. But you can also do more complicated table processing, like renaming columns.

TAGS

One more slice of Cucumber I found convenient for this project is the notion of tagging tests as belonging to different categories.

```
Feature: Stations
  ...

  @restart
  Scenario: Remembering stations
    Given a list of radio stations "KBOO,KINK"
    And a test server
    When I restart the app
    Then I should see the stations "KBOO,KINK"
```

For instance... Apple’s Instruments tool doesn’t like it when your app exits. So I tagged all the scenarios that required a restart.

```
cucumber -t~restart features
```

When you run your tests, you can tell Cucumber to include or exclude particular tags.

RESULTS

For an investment of about 10% of the project’s time maintaining the test infrastructure, using executable examples served to: 1) drive out the design, 2) catch a couple of minor bugs, 3) help replicate other bugs found manually, and 4) find leaks (mine and others’). Plus, now I know what’s on the radio.

} /undees/justplayed

The app should be coming out Real Soon Now (tm) for free on the App Store, but you can get the code today. It’s on the ‘Hub and in the ‘Bucket. If you want to deploy to a real iPhone, you’ll need a $99 license from Apple—so I guess until they accept and publish the app, it’s free as in “speech,” but not free as in “beer.”

CONVERSATION

- Barnes and Noble booth
  During lunch today
- Meet Authors from the Pragmatic Bookshelf
  7 PM tonight, Ballroom A7
- undees@gmail.com
- http://twitter.com/undees

If you’d like to continue the conversation, catch me in the hall, or come to one of the scheduled events.

SHOULDERS OF GIANTS

- Bromine
- cocoahttpserver
- ASIHTTPRequest
- hg-git
- Pivotal Tracker
- iBetaTest
- api.yes.com

Special thanks to yes.com for making their radio playlist API available under a Creative Commons license.

CREDITS

- Ruby icon (c) Iconshock
- Instrument photos (c) Tektronix
- Squirrel (cc) kthypryn on Flickr
- Hand (cc) myklroventine on Flickr
- Logos (tm) github and bitbucket

Writing the software and preparing the material for this talk has been a blast. I can’t wait to see what the rest of OSCON has in store for us, and am grateful for our time together this week. Cheers.

BONUS SLIDES

Here are a few more tidbits that didn't make it into the main presentation.

UNIT TESTING

RSpec and iPhone—Dr. Nic Williams
http://drnicwilliams.com/2008/07/04/unit-testing-iphone-apps-with-ruby-rbiphonetest/

I experimented a little with using Dr. Nic's rbiphone library to unit-test my Objective-C model classes from Ruby using the RSpec framework. This technique wasn't a perfect fit for the models in my app, which were so tied to Apple APIs that there wasn't much for the tests to do.

SCREEN CAPTURE

Google Toolbox for Mac

http://code.google.com/p/google-toolbox-for-mac/wiki/iPhoneUnitTesting

Google has a suite of Mac and iPhone development tools. Their iPhone unit test code is geared toward capturing screen images and comparing them against a reference. Heavily graphical apps, or apps that don't use the standard UIKit controls, spring to mind for a technique like this.