Extracted from:

Test iOS Apps with UI Automation

Bug Hunting Made Easy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Test iOS Apps with UI Automation

Bug Hunting Made Easy

Jonathan Penn

Edited by Brian P. Hogan

Test iOS Apps with UI Automation

Bug Hunting Made Easy

Jonathan Penn

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-52-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2013

6.3 Building an iPad Test Suite with Reusable Pieces

We've seen how the element tree can vary on the iPad in different orientations. Now we want to apply that knowledge to our test scripts. How much can we reuse? What should we separate?

Because the interfaces are so different between the devices, it is best to make a separate test-suite file for the iPad. We can then pull in the screen objects one by one to figure out what can be reused. Here we're going to write scripts to detect the device model and orientation and then use these to help the SearchTermScreen object find the elements it needs to do its job.

Finding the Device Model

The UIATarget instance provides the model() method that returns a string describing what kind of system the app is running on. Let's run this small script to see what it returns for us now:

06-Universal/step05/automation/sandbox.js

```
var target = UIATarget.localTarget();
➤ UIALogger.logMessage(target.model());
```

In our case, we see iPad Simulator. Let's write up a quick script to test for this by using a JavaScript substring search:

06-Universal/step06/automation/sandbox.js

```
var target = UIATarget.localTarget();
if (target.model().match("iPad")) {
    UIALogger.logMessage("Running on iPad");
} else {
    UIALogger.logMessage("Running on iPhone/iPod Touch");
}
```

The match() method will return true if the string contains iPad. Since we'll be testing for this condition a lot we should pull it out into a reusable function, but where do we put it? If it were a tool that affected our test environment, then we could put it in the global namespace alongside test(), log(), and all the rest. If this affected a portion of the screen, we could put it in the appropriate screen object. Instead, this kind of method relates more to the application as a whole. So, let's create an object that reflects that and put it in lib/App.js:

06-Universal/step07/automation/lib/App.js

```
"use strict";

var App = {
    isOnIPad: function() {
        return this.target().model().match("iPad");
    },
    target: function() {
        return UIATarget.localTarget();
    }
};
```

For those of you who have strong opinions on how to camel-case the iDevice names, please cut me some slack as I name this function, isOnIPad(). I spent way too long trying to decide how to do it. Substitute your own treatment to taste.

To use this new object, we need to import the App.js file in our test-environment file so it is available everywhere we need it:

06-Universal/step07/automation/env.js
```
#import "lib/App.js";
```

We'll use this App object as a nice namespace to keep our application-level concerns separate from our other tools. In this case, it's as if we're asking the application, "Are you running on the iPad?"

Testing in Landscape Orientation

Now that we have a way to determine the device family, we are ready to start forking test code in strategic places. In an ideal world, it would be great to run a single test suite that adapts itself completely to any idiom. In practice, the iPad interface is so unique that we benefit from building a separate test suite composed of common pieces instead.

Let's create a new script file named automation/test_suite_ipad.js that imports our test environment so we have all the tools and shortcuts we started building in Chapter 4, *Organizing Test Code*, on page ?:

06-Universal/step08/automation/test_suite_ipad.js
```
"use strict";
#import "env.js";
if (!App.isOnIPad()) {
    throw new Error("Test suite only works on iPad");
}
```

As a courtesy to ourselves, we're also adding a quick conditional check that throws an error if this suite isn't run against an iPad. Now let's write a test that removes and then replaces the "coffee" search term in landscape orientation:

06-Universal/step08/automation/test_suite_ipad.js
```
test("Removing and replacing search term in landscape", function() {
➤   App.rotateLandscape();
    var s = SearchTermScreen;
    s.removeTerm("coffee");
    s.assertNoTerm("coffee");
    s.addTerm('coffee');
    s.assertTerm(0, 'coffee');
});
```

Previously, we had this broken out into two separate behavior tests when we experimented with building this up for the iPhone. In practice, you can mix and match your test steps any way that makes sense to you. In our case, this is just how the iPhone tests emerged. Since we're at a different starting point with the iPad, let's combine the removing and replacing behavior into one test since it represents a complete thought that puts the application back in the starting state.

We're testing landscape orientation first because we know the element tree is close enough to the iPhone version that this test should pass. The SearchTermScreen object looks for the first navigation bar and table view in the window to do its work. It should have no trouble navigating the split view controller in landscape orientation.

Note the call to the App.rotateLandscape() method. We haven't written that yet, so let's do so now:

06-Universal/step08/automation/lib/App.js

```
var App = {
    // ...
    rotateLandscape: function() {
        var orientation = UIA_DEVICE_ORIENTATION_LANDSCAPELEFT;
        this.target().setDeviceOrientation(orientation);
    },
    // ...
};
```

This gives us a meaningful yet terse way to rotate the device to the other orientation that we need to support.

Run the test suite; it passes. There's nothing like a quick and easy win to pump up our motivation before we climb the challenging hill.

Testing in Portrait Orientation

Things get tricky when we want to test in portrait orientation. Add this script to the end of the test-suite file we just started to see what happens:

06-Universal/step09/automation/test_suite_ipad.js

```
test("Removing and replacing search term in portrait", function() {
➤   App.rotatePortrait();
    var s = SearchTermScreen;
    s.removeTerm("coffee");
    s.assertNoTerm("coffee");
    s.addTerm('coffee');
    s.assertTerm(0, 'coffee');
});
```

It's almost exactly the same code as the other test, but instead we're starting in portrait orientation. This is an example where some code duplication can be useful. We've pulled the actual steps for removing, adding, and asserting search terms into the SearchTermScreen object and given them meaningful names. Repeating these names here helps us understand what we're actually testing while giving us the ability to reuse test code. Of course, you can take test-code reuse too far, but you get the point. Readability is the guiding factor, and deciding between duplication and extraction for reuse is a pragmatic matter.

Before we run this, we need to implement the method on the App object to rotate to portrait orientation:

06-Universal/step09/automation/lib/App.js

```
var App = {
    // ...
    rotatePortrait: function() {
        var orientation = UIA_DEVICE_ORIENTATION_PORTRAIT;
        this.target().setDeviceOrientation(orientation);
    },
    // ...
};
```

When run, we're greeted with a passing first test and a failing second test, as we expect. Our SearchTermScreen object doesn't yet know about the popover element. Let's fix that.

Adapting Screen Objects to New Idioms

We have a series of changes to make to fix our SearchTermScreen object so it works in portrait orientation. Let's look at the first point of failure in our removeTerm() method:

06-Universal/step09/automation/lib/screens/SearchTermScreen.js

```
// ...
removeTerm: function(name) {
    log("Removing search term", name);
➤   var editButton = this.navigationBar().leftButton();
    editButton.tap();
    var tableView = this.window().tableViews()[0];
    var cell = tableView.cells()[name];
    var deleteSwitch = cell.switches()[0];
    deleteSwitch.tap();
    var deleteButton = cell.buttons()[0];
    deleteButton.tap();
    editButton.tap();
},
// ...
```

The failure happens when we try to call this.navigationBar(), which has the Screen prototype fetch the navigation bar for us. We need to change how we fetch the navigation bar. Thanks to the JavaScript object model, we can override the navigationBar() method only in the SearchTermScreen object:

06-Universal/step10/automation/lib/screens/SearchTermScreen.js

```
navigationBar: function() {
    if (App.isOnIPad()) {
        return this.window().popover().navigationBar();
    } else {
        return this.__proto__.navigationBar();
    }
}
```

Now, when we call the navigationBar() method, this local version checks to see if we're on the iPad and then fetches the navigation bar from within the window's popover element. If we're not on the iPad, we return the original value by calling navigationBar() on the prototype, which is the JavaScript approximation of sending a message to super in Objective-C.

We have a problem, though. The landscape version of the test is now broken since we're always assuming there's a popover if running on the iPad. We need to add another method on the App object to check for orientation:

06-Universal/step11/automation/lib/App.js

```
var App = {
    // ...

    isPortrait: function() {
        var orientation = this.target().deviceOrientation();
        return orientation == UIA_DEVICE_ORIENTATION_PORTRAIT ||
            orientation == UIA_DEVICE_ORIENTATION_PORTRAIT_UPSIDEDOWN;
    },

    // ...
};
```

And now we can fix our conditional expression to look for the popover only when in portrait orientation on the iPad:

06-Universal/step11/automation/lib/screens/SearchTermScreen.js

```
➤ if (App.isOnIPad() && App.isPortrait()) {
      return this.window().popover().navigationBar();
  } else {
      return this.__proto__.navigationBar();
}
```

Let's look at the rest of the removeTerm() method to see what else we may need to fix:

06-Universal/step12/automation/lib/screens/SearchTermScreen.js

```
removeTerm: function(name) {
    log("Removing search term", name);
    var editButton = this.navigationBar().leftButton();
    editButton.tap();

➤   var tableView = this.window().tableViews()[0];
    var cell = tableView.cells()[name];

    var deleteSwitch = cell.switches()[0];
    deleteSwitch.tap();

    var deleteButton = cell.buttons()[0];
    deleteButton.tap();

    editButton.tap();
},
```

Ah, the table view. Just like the navigation bar, it's no longer a direct child of the window. We're fetching it and saving it in a local variable in several places in this screen object. This is a great time to pull it out into an instance method:

06-Universal/step13/automation/lib/screens/SearchTermScreen.js

```
tableView: function() {
    return this.window().tableViews()[0];
}
```

Now we need to change every place where we fetch the table view from the window to use this method instead:

06-Universal/step13/automation/lib/screens/SearchTermScreen.js

```
// ...
➤ var cell = this.tableView().cells()[name];

var deleteSwitch = cell.switches()[0];
deleteSwitch.tap();
// ...
```

At this point we should run the iPhone test suite against the iPhone simulator to make sure we didn't break anything. We're taking a small redesign step here by pulling out the tableView() method and checking our work. It's tempting to just plow through and make the changes we're intending to make, but taking small, executable steps is very important for a dynamically typed language like JavaScript.

Once we're confident that the test code still functions as it did before, we can augment the tableView() method to take the device and orientation into account:

06-Universal/step14/automation/lib/screens/SearchTermScreen.js

```
tableView: function() {
    var root;
    if (App.isOnIPad() && App.isPortrait()) {
        root = this.window().popover();
    } else {
        root = this.window();
    }
    return root.tableViews()[0];
}
```

This time, instead of delegating to the prototype in the else clause, we're using a variable named root to hold the root of the element tree for this screen object's elements. Sometimes it can be useful to think in terms of element subtrees. If we're in portrait orientation on the iPad, we want to find the table view element in the popover's subtree; otherwise, we want to find it in the window.

Finally, our portrait test is ready. Run it and watch both orientations pass the tests with flying colors. We've successfully adapted our reusable test-script pieces for these two tests. Run the iPhone test suite against the iPhone simulator, and we'll pass there, too.

We've walked through some steps to surgically adapt our screen objects to changes in the element tree. This is yet another advantage of grouping and organizing test steps by parts of the screen. Where appropriate, we can give these objects the intelligence they need to keep working in a variety of conditions.

Sometimes, though, we know exactly what we're looking for on the screen, and we just want a quick and dirty way to find it. Next we'll discuss how to recursively search the entire tree with predicate expressions.