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:

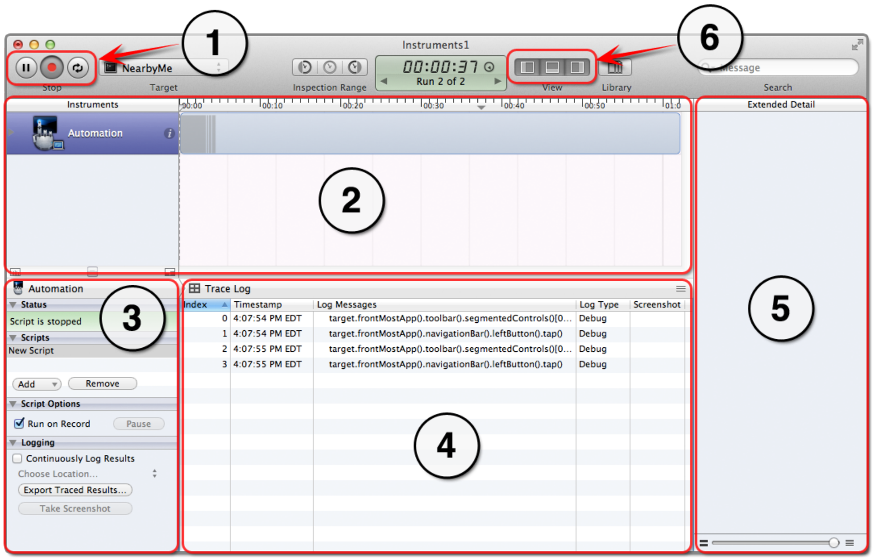
Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

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

1.2 Finding Our Way around UI Automation

At first glance, UI Automation and Instruments together can be pretty overwhelming. In this section, we're going to take a step back and examine what we've been using. Instruments is primarily a profiling tool built on the idea of running and recording a *trace* of activity in an application.

The window we've been working in is a *trace document* where all the activities over the course of the application run are traced. The key components of the trace-document window are broken down in the following figure.



```
Index  Timestamp       Log Messages                                           Log Type  Screenshot
    0  4:07:54 PM EDT  target.frontMostApp().toolbar().segmentedControls()[0...  Debug
    1  4:07:54 PM EDT  target.frontMostApp().navigationBar().leftButton().tap()  Debug
    2  4:07:55 PM EDT  target.frontMostApp().toolbar().segmentedControls()[0...  Debug
    3  4:07:55 PM EDT  target.frontMostApp().navigationBar().leftButton().tap()  Debug
```

Figure 7—Getting to know Instruments

1. These buttons control the *trace recording* in Instruments. The red record button is the one we'll use the most. When it's pressed, Instruments runs and attaches to the target application and starts recording the trace. When it's pressed again, the application is terminated and the trace recording stops. Note that this is *not* the same as the red record button at the bottom of the script pane we used to capture a script.

2. This *track pane* is where all the instruments show up and give a graphical overview of what they observe during the trace. Right now we have only the UI Automation instrument, and we'll see green or red regions in the timeline depending on whether the tests pass or fail.

3. We use the left sidebar to command and control the UI Automation instrument. The Status section shows us the state of the automator, currently stopped. The Scripts section is where we manage sets of scripts to run inline. The Script Options area gives us execution control, like if we should automatically run a script when a trace recording begins. And the Logging section has to do with the automation trace logs that we'll get to later.

4. This lower pane is the heart of the automation instrument. Every time we play back an automation script, the pane switches to show the trace log, as we see here. We can switch back to the script pane by clicking the pop-up button named Trace Log at the top of the pane and choosing Script from the menu. There's also an undocumented shortcut to toggle between the script and trace-log panes by double-clicking on the text in the Status section of the left sidebar.

5. The right sidebar is where we'll see more details for individual items selected in the trace log. JavaScript exceptions or usage errors also show more detail here.

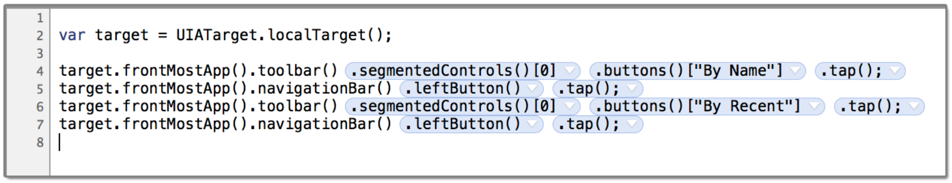
6. These toolbar buttons let us toggle the visibility of each of these panes. We can turn the sidebars off when we need more room.

Tweaking the Captured Script

Now let's dissect the script that came out of the recording; we'll switch back to the script pane so we see what's in Figure 8, *The recorder's initial output,* on page 3.

The captured automation script starts with the `target` object that represents the simulator, gets the frontmost application, and then drills down into the interface to identify which button will be tapped.

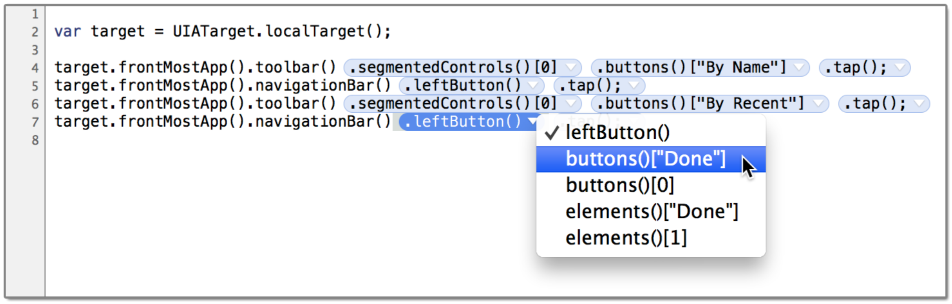
A blue bubble means that there is more than one way to reference that element on the screen. UI Automation picked one that it thought was most suitable while capturing our actions, but we can click the disclosure triangle to choose a more meaningful one if we want. For example, we can change the reference to the left-hand button in the navigation bar to instead look up the button by the name Done, as the following figure shows.



```
var target = UIATarget.localTarget();

target.frontMostApp().toolbar() .segmentedControls()[0] .buttons()["By Name"] .tap();
target.frontMostApp().navigationBar() .leftButton() .tap();
target.frontMostApp().toolbar() .segmentedControls()[0] .buttons()["By Recent"] .tap();
target.frontMostApp().navigationBar() .leftButton() .tap();
```

Figure 8—The recorder's initial output



```
var target = UIATarget.localTarget();

target.frontMostApp().toolbar() .segmentedControls()[0] .buttons()["By Name"] .tap();
target.frontMostApp().navigationBar() .leftButton() .tap();
target.frontMostApp().toolbar() .segmentedControls()[0] .buttons()["By Recent"] .tap();
target.frontMostApp().navigationBar() .leftButton()
```

Figure 9—Choosing the table view by index

Capturing scripts like this is a great way to practice exploring this scripting interface. If you're ever stumped on how to reach for a certain control or you want to see other ways to do it, capturing your actions can help.

Limitations of Capturing Actions

Unfortunately, the script-capturing mechanism has limitations, too. Sometimes it can have trouble finding buttons on the screen, or it can get confused when trying to record raw gesture events.

For example, we'll try tapping the + button in the app to add a new search term to the list. When we tap on it, an alert will pop up with a text field and show the onscreen keyboard (as the following figure shows). Let's capture these steps and see what happens.

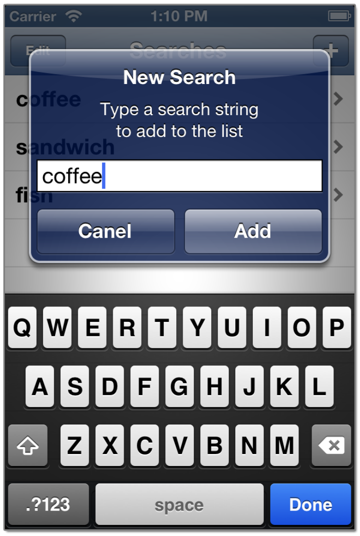
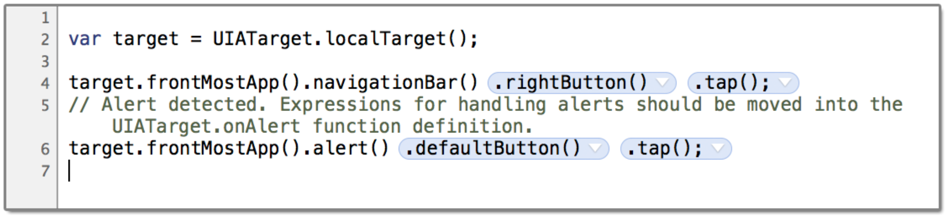


Figure 10—Capturing actions involving alerts

Create a new script by choosing Add > Create in the script sidebar. Press the red record button at the bottom of the script pane, switch to iOS Simulator, and then tap the + button in NearbyMe's navigation bar. On your Mac keyboard, type the word coffee and then tap Add to create the search term. In Instruments, press the stop button beneath the script pane to see this captured script (shown in the figure here).



```
var target = UIATarget.localTarget();

target.frontMostApp().navigationBar() .rightButton() .tap();
// Alert detected. Expressions for handling alerts should be moved into the
    UIATarget.onAlert function definition.
target.frontMostApp().alert() .defaultButton() .tap();
```

Figure 11—Confusing the UI Automation script-capture mechanism

UI Automation balked when faced with the alert. We'll have to step in and do our own manual work to handle the alert and decide what to do. That can wait until Section 3.1, *Testing with a Modal Alert View*, on page ?. For the moment, just know that the capturing process isn't recording raw data for your *exact* steps and your timing as you perform them. It is trying to convert what you are doing into individual automation-script lines.

Beyond the Basics

Phew! That was a whirlwind tour of UI Automation and Instruments. We recorded some actions and stepped back to examine how we did it and what it produced. Try these techniques to capture and play back your actions as you poke around in your applications. What happens if you try to capture gestures such as swipes or pinches? You'll undoubtedly run into limitations like what we experienced here, but it's a great place to start and learn.

Capturing events can be a useful way to get up and running quickly, but if we're going to build long-living and quickly adapted test scripts, we must learn to write our own by hand. To take our UI Automation scripting to the next level, let's dig into the language and interface.