What Your UI Tests Need to Say

Joe Ferrara

Abstract

UI tests that only execute steps and return a Pass or a Fail are not very useful. While this kind of automation gets you much further along in your automation, it requires close examination of the test code and its logs to figure out what went wrong. One of the best practices you will learn is that the test should communicate what it is doing plus provide a precise statement of what went wrong. Other best practices having to do with test results communication are reporting results to a test case management system and communicating to stakeholders on the status of test runs. Aspects of test reporting such as the different audiences that care about test report data will be covered. Audiences range from the individual engineer to senior management.

In this talk, I will use code from an iOS UI tests project to illustrate fine grained ways that tests communicate progress and status, such as ways to use assertions, screenshots and other test artifacts to your advantage. I will also describe a component written in Swift that that writes test results to a test case management system, TestRail. The best practices and experiences shared are applicable to software platforms besides iOS.

Biography

I am a Staff Engineer, working in the front-end test architect role, at The Climate Corporation. I have more than 20 years of experience ranging from development, test and test development for numerous companies. I have led the development of UI test automation projects on several platforms. I spent over 10 years at Microsoft developing consumer multimedia software, testing and automating tests for consumer and online software. I spent time at startups in Seattle plus consulted with several companies including White Pages and Starbucks. My focus for the past several years has been mobile application testing, mainly on the iOS platform.

1 Introduction

Effective communication from your automated tests to members of your team is important for any team process and for continuous integration and continuous deployment (CI/CD) pipelines. With effective communication, issues can be investigated quickly and understood efficiently. Whether you are part of a large test engineering team or just one or two people, your time is valuable. Being able to zero in on an issue quickly frees up your time for testing and development.

In this paper, I illustrate the best practices that I have worked through in my work with UI test automation development. My most recent experience is with testing iOS apps using the Swift language. My examples are based on this platform, though the material is applicable to other toolchains. I first lay out the reasons why communication is important and then get into specifics applicable to authoring the test function in code and communication to individual engineers, the test case database and to any member of your team.

I have worked on teams where the choice of a UI test framework allows for writing tests in the same language that the development team codes with. This decision allows the test engineering team to fully engage the development team in reading, writing and maintaining UI tests. The test framework I currently use, XCUITest, is included in Apple's Xcode along with the unit test framework, XCTest. The benefit for my projects has been that work can be leveraged between dev and test. I have worked with others to write regression tests that are required for an effective automated CI/CD pipeline. I have seen developers also use UI tests to automate bugs and use those automated test cases to verify bugs that get fixed.

2 Preparing for Writing Tests

2.1 When to Write UI tests

There are at least three phases of a project where UI tests can be written. One is before the feature is written. This is difficult with a test that manipulates a UI, but it is not impossible. At least outline in high level code what is supposed to happen and failing by default.

A second phase is while the features are being implemented. Working with the app developers can ensure that the automation is buildable and gives early feedback to stakeholders.

A third phase is to automate an existing set of tests for features that have already shipped. A motivation for doing so is to cut down on the time required for manual testing and to enable the continuous integration pipeline to run faster and with fewer manual resources.

2.2 What to Communicate When a Test Fails

When writing a test, it is tempting to do whatever it takes to write code that passes the given test case or specification you have for the test. This is ultimately what you want, but it is also important to keep in mind what happens when the test fails. A future "you" or anyone else on your team may need to deal with this.

In addition to cases when the application under test fails, your test can fail due to the runtime environment, an error in the test code, a change in the underlying test framework or change in the operating system.

When planning for implementing tests you have to specify for yourself how your tests will communicate. This includes assertion messaging, logging, screen shots and text attachments. You can build some of the enabling functionality into your test framework. For example, in a recent iOS automation project, I implemented code in the common **tearDown()** logic so that a snapshot of the UI element tree is saved to a text file attachment that is always available as a test artifact. For logging, I wrapped the most basic "print" function with one which can direct where text is sent so as to accommodate a cloud testing framework.

3 Aspects of Assertion Messages

3.1 Assertions

Assertions constitute the core of your communication. Other than simply seeing that a test failed, the assertion message will often zero in on what the failure is. To the extent possible, providing enough information to get to a root cause is ideal. The assertion logic and messaging work in concert with additional test artifacts like screen shots, attachments and logging.

The assertion functions provided in XCTest are **XCTAssertTrue()**, **XCTAssertFalse()**, **XCTAssertNil()**, **XCTAssertNotNil()**, **XCTAssertEqual()**, **XCTAssertNotEqual()**, **XCTAssertThrowsError()**, **XCTAssertNoThrow()**, **XCTAssertGreaterThan()**, **XCTAssertGreaterThanOrEqual()**, **XCTAssertLessThat()**, **XCTAssertLessThanOrEqual()** AND **XCTFail()**. Each assertion function takes an optional parameter for a message.

XCTAssertTrue() could be used for most types of assertions, however in that case the intent is more difficult to understand from reading the code and you lose any default messaging the test framework gives you. Of the assertions shown here for XCTest, only the ones having to do with NSExceptions need some specific assertion function. Although the other specialized assertions can be replaced with **XCTAssertTrue()**, this is not recommended.

I often see code that does something like 'XCTAssertTrue(!someFlag, "something went wrong")'. Here, the thing that might get lost in a quick scan of the code is the exclamation operator just to left of `someFlag`. It is better to just use **XCTAssertFalse()** instead, so that the intent is very clear just from reading the function name.

XCTAssertEqual() and **XCTAssertNotEqual()** are examples where the assertion function provides for a default message before whatever message is specified in the parameter. It is much better to be explicit that you are testing for a comparison so that when it fails the XCTest framework it will include the compared values in the message.

3.2 Messages for Assertions

3.2.1 Do Include a Message

I have seen engineers write assertions in UI tests that lack any message. I have found that when I get a failed test case with no error that I am, at first, at a loss as to what is going on. I am forced to go into the code and tease out what is being checked and what happens before the assertion is fired. Going into the extra steps is certainly not impossible and might only add a few minutes to writing a bug report or determining that one is not needed. However, why waste time when a clearer communication is easier to do?

3.2.2 Include Context

No matter what type of assertion is used, it is good to include additional data in the message that include the names of the elements being compared -- additional information on what precedes the assertion. I have sometimes had the opportunity to add a hint on what could be going wrong. For example, if some setup is required in a test and lack of doing that setup fires off an assertion, then why not inform the reader about the setup requirement?

In addition to being aware of coding guidelines while writing code, code reviews are a great opportunity to catch cases where context is not being provided. Some teams create short guidelines for code reviews. From a test engineering point of view, it is valuable to advocate for these sorts of guidelines and participate in writing them.

3.2.3 Consider Custom Assertions

XCTest and many other frameworks will allow for some method to write custom assertions. This allows the developer to write very specific verifications particular to their application. Doing so can often eliminate duplicate code and make it easier to both comprehend and maintain.

When you see many assertions that look similar and do something special before each call, you have a candidate for a custom assertion. For example, I want to assert that a substring shows up within a string.

Below is an example for code that will benefit from a custom assertion.

```
let myList = ["apple", "orange"]
var searchFor = "apple"

XCTAssertTrue(myList.contains(searchFor),
    "Mylist: \(myList) does not contain \(searchFor)")
searchFor = "orange"

XCTAssertTrue(myList.contains(searchFor),
    "Mylist: \(myList) does not contain \(searchFor)")
searchFor = "banana"

XCTAssertTrue(myList.contains(searchFor),
    "Mylist: \(myList) does not contain \(searchFor)")
```

Below is the custom assertion that can be used instead of **XCAssertTrue()**. The function usage is shown first.

```
CXCTAssertListContains(myList, searchFor)

extension XCTest {
    func CXTCAssertListContains(list: [String], searchFor: String,
        message: String = "", file: StaticString =  #file,
            line: UInt = #line) {
    let errorMessage = (message == "") ?
            "List `\(list)` does not contain `\(searchFor)`" : message
     XCTAssertTrue(list.contains(searchFor), errorMessage,
            file: file, line: line)
    }
}
```

When naming your custom assertion, it is clearest to not use the same prefix as the test framework. So, in the example, instead of using `XCT` use `CXCT` where `C` stands for Custom.

4 Communicating to the Engineer

4.1 Organizing Test Case into Steps

In XCTest there is a function, **runActivity()**, which is used with a named closure to show test code in a distinct step. That name shows up in the report's hierarchy making the report easier to comprehend.

Sample Code

```
XCTContext.runActivity(named: "Verify starting state") { _ in
  XCTAssertTrue(demoScreen.labelStaticText.exists,
        "Demo Label element is missing")
  XCTAssertTrue(demoScreen.labelStaticText.label == "",
        "Demo Label is not empty")
}
```

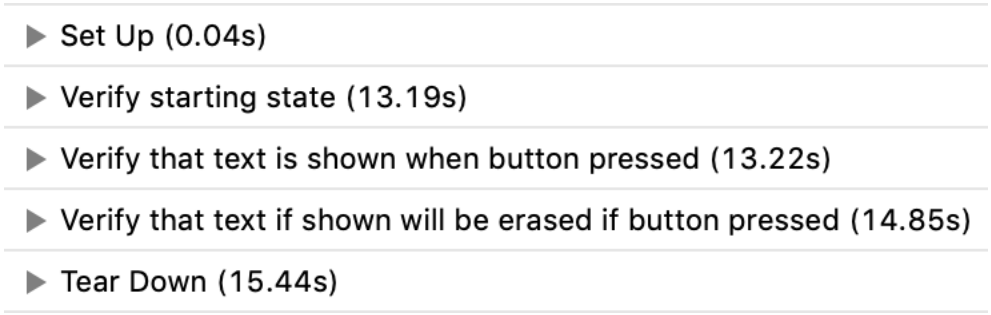

Figure 1 - Test Report Showing Activities

4.2 Screen Shots

4.2.1 Screenshots Created Automatically

In XCUITest, the system framework automatically takes screenshots whenever there is some change in the state of the app. I find these useful, though sometimes it is a bit time consuming to match up which automatic screen shots to pay attention to. It is also sometimes the case that when the test fails, the most relevant screen shot is not the last one.

4.2.2 Screenshots Created in Code

Adding code to a test case that saves a screen shot as an attachment allows for the test author to specify a name or to even do something like take an image of some subset of the screen. A reason for doing this rather than only relying on automatic screen shots is that if some UI state more than one step before an error is related to the screen where the error occurs. For example, in an app that has a form for entering data the test could modify data and then navigate through a few other screens before landing on one where it needs to assert that the data is displayed correctly. By keeping a named, manually attached screen shot, it is easier to make a comparison between what the test actually entered in the new value and the later screen where it displays the new value.

An example for adding an attachment in code is shown below.

```
let screenshot = app.windows.firstMatch.screenshot()
let attachment = XCTAttachment(screenshot: screenshot)
attachment.lifetime = .keepAlways
attachment.name = "Enter new value in `Name` field"
add(attachment)
```

4.2.3 Screenshots Retention

Screenshots can be useful even when the test passes. These can be configured to persist for all test runs. I find these useful when a test is inconsistently failing or when I am trying to compare an earlier integration's passing result to the current integration's passing result. It is typically possible for a CI/CD pipeline to have the capability of deleting these artifacts after some amount of time has passed.

4.3 Keeping Key Screenshots

Some tests will fail with some data verification from screen to screen. It would be useful to not only have detailed data on the final screen that is up when the assertion fires but to also have the first screen with the data that was compared against. By naming these screen shots and having them readily available for failures, it helps when investigating the issue.

4.4 Code Clarity

4.4.1 Accessibility Identifiers

Accessibility identifiers are more of a coding detail, though the clarity of your code makes it easier to have coherent assertion messages and diagnosis support. In iOS apps, the developer can associate an accessibility identifier with any element on the screen. There are a few ways to do this though demonstrating these is out of the scope for this paper.

Having a naming convention is important. One convention is to start with a name for the screen, then the name of the element and then the name for the type of element. For example, a name for the user name text field on a Sign In screen is **SignIn_Username_TextField** is useful.

How you deal with system elements such as those in iOS itself is important. For these you could be forced to use labels or position in the element tree. One work-around is to copy the localization string into a place accessible by the test framework and then use a utility function to get labels using an internal identifier.

5 Communicating to the Test Case Management System

5.1 Design

The Test Case Management system used on my projects is Gurock Software's TestRail tool. It is a server-based service that stores a team's test cases, test plans and test results. It is a useful place to store documentation for test cases. It supports creating detailed reports based on past test runs. One key scenario I have an encountered is a way to write results from UI tests automation runs to TestRail. The REST endpoint necessary is, "api/v2/add_result_for_case", in the TestRail instance.

An executed test result in a TestRail instance can be set to Pass, Fail, Untested, Retest, or Blocked. Custom attributes often defined in a mobile device-oriented system to store build label, comment, device and operating system. The build label and comment are passed into the reporter component. The device and operating system data can be determined in code at runtime. When passing data to the TestRail client, include an API key and test Run ID. The API key is used instead of account and password in code to improve security. The Run ID is needed by TestRail so it knows where to write results.

The use case described here is for using the TestRail API was simply to record basic results from UI tests. UI automation writes Pass or Fail results to TestRail for each test case using a REST endpoint. A test case where a result is not written remains Untested by default. Retry and most Blocked test case results are not supported with automation but used manually. If a test result is set to Nil then the test result is set to Blocked. Each result also includes the meta data supplied by the TestRail client and system derived data. Each test result is made for the correct test case since that ID is also passed in.

In each test case an instance variable representing the test case ID string representing one or more test case IDs. If the variable contains a comma delimited list of IDs then the results get written to multiple test cases. It is my recommendation that a class instance method is the preferred way to store a test ID so as to hide variables from the test case and make it easier to make changes. The sample function signatures below support setting a single ID or an array. You could consider validating the string form in the function body and returning a Boolean result or throwing an error. See below.

```
func setTestCaseId(id: String)
func setTestCaseId(id: String[])
```

5.2 Implementation Summary

In the References section at the end of this paper, you will find a link to my blog that contains some more detail about how to write UI Test results to TestRail.

In the test's teardown function, one of the steps is to set the test case result based on data about the test run. XCUITest has a way to find out how many tests failed. A **testRun** variable from the XCUITest framework has an optional value, **testRun?.failureCount**. If the failure count is greater than 0 then you know that one or more tests failed. A function with a name like **reportResultsToTestRail()** can then be called.

The test case ID string must be parsed so that an ID can be included with each post to TestRail. The next step is to configure the test data that gets passed into the test case including an API key which is used for authorization into your TestRail instance. Other data included is for App version, Bot name, Bot integration ID, Host name, Comment, TestRail account name, and TestRail API key.

The data that gets collected is then transformed into a JSON object which can be passed into a Swift API that posts the data to the TestRail endpoint.

Below is the code for setting the endpoint's URL.

```
let postUrl = "https://mycompany.TestRail.com/index.php?/api/v2/add_result_for_case/" + testRunId + "/"
+ testCaseId
```

Any errors are determined and logged. Data from errors should at least include an HTTP status code but can also include full details from the call's response. If an error occurs the TestRail result is left as its default value of Untested. Further investigation should proceed using logged data.

6 Communicating to the Team

Two routes that come up at this level of communication are summaries of test status as well as input to an issue tracking system where test execution data is communicated to all team members.

Test results flow through a CI/CD pipeline. Jenkins and Gitlab are a couple of tools that are used for supporting CI/CD.

A summary of details about the executed tests help to keep management apprised of the health of the software and its readiness for shipping. Information that should be included for high level reporting include which tests fail, pass, and not tested. A couple of additional characteristics are revealing which test cases require another run or those that are blocked. These additional characteristics are most commonly associated with manual test results.

Tools such as TestRail provide for reporting capabilities that can be used for visualizations such as a pie chart. Another component of TestRail is more advanced reporting which can combine test status with other properties defined for each test case such as its type, priority and severity.

For test results dashboards and reports, assertions messages and other lower level details are not commonly found as it is possible to drill down into details through the TestRail result and then the integration results.

6.1 Reports

Reports should include the following information. Percentages by test status in conjunction with a pie chart visually makes clear how many failures occurred as well as how many test cases passed. Something that can fit on one page is useful for sending to the team daily to provide a bird's eye view of the application's health. A high-level report can also be incorporated into a dashboard that some team members will use to get some more detail. This dashboard can dig deeper into the data and show the history for test status from build to build as well as a history of the specific test cases that fail and issues created as a result of test failures. Exploring this view gives the reader a sense of the application's stability over time.

6.2 Issue Tickets

Information communicated by UI tests in conjunction with artifacts from the build process and pipeline steps is needed for the creation of issue tracking tickets. Tickets are created either manually or through automation found in the CI/CD pipeline. A detailed look at aspects of issue tracking is beyond the scope of this paper, but several key things are worth noting.

One tool for tracking application issues is an Atlassian product, Jira. Reporting done with Jira can reveal the number of issues found over time, each issue’s persistence over a number of builds, and a measure of test flakiness. The data associated with a test failure is useful for the contents of a ticket. Artifacts such as assertion messages, logs, attachments, and a reference to the test case management system are also important.

Deciding what data to communicate and how to do so is crucial for forming actionable issue tickets and facilitating ease of analysis by the engineer. Issue tickets are typically triaged manually by a group of team members that includes test engineering, development, and product. The information that your test communicates into the ticket should be clearly stated so that a diverse group of team members can understand the issue quickly and then make appropriate decisions about next steps. Another team member who benefits from effective communication is either yourself or someone else who needs to

manually reproduce an error reported from tests. This helps with testing around the issue and determining if it can be reproduced. It is also at this point that test flakiness can be revealed.

7 Conclusion

Improving what your UI tests say supports faster issue diagnosis and better decisions from team members. Team members other than the test case coder benefit from high quality communication so that a clear picture is formed that is not missing details. Other team members who may not have been involved or remember details of a test or its feature under test need that information to make good decisions. Appropriate assertions and a precise message to indicate what has gone wrong is essential.

When communicating to the Engineer, you have the responsibility to include precise details, context and intent. Precise details come into play when constructing assertion messages that include state and relevant data. Context is adding any information about what leads up to the assertion. Intent is found in the code when choosing an appropriate assertion function to use and how the test code is organized with an eye towards how the test results will be displayed when test execution is done.

When communicating to the test case management system, you have the responsibility to forward test status, application metadata such as build label, device, and OS. Assertion messages are also useful.

When communicating to the team, you have the responsibility to provide clear information concisely using data derived from lower level communication. The sum of all the levels of communications provides enough detail to write actionable issue tickets.

8 Resources

Deeper Dive into Reporting Results to TestRail - https://joeferrara24.wordpress.com/2019/08/30/reporting-ui-test-results-to-testrail/

GitLab - https://about.gitlab.com

Jenkins - https://jenkins.io

Jira – https://www.atlassian.com/software/jira

TestRail – https://www.gurock.com/testrail

TestsRail API - http://docs.gurock.com/testrail-api2/start

XCTest documentation - https://developer.apple.com/documentation/xctest

XCUITest User Interface Tests - https://developer.apple.com/documentation/xctest/user_interface_tests