Testing Basics

A test is code you write that exercises your app and library code and results in a pass or fail result, measured against a set of expectations. A test might check the state of an object's instance variables after performing some operations, verify that your code throws a particular exception when subjected to boundary conditions, and so forth. For a performance measuring test, the reference standard could be a maximum amount of time within which you expect a set of routines to run to completion.

Defining Test Scope

All software is built using composition; that is, smaller components are arranged together to form larger, higher-level components with greater functionality until the goals and requirements of the project are met. Good testing practice is to have tests that cover functionality at all levels of this composition. XCTest allows you to write tests for components at any level.

It's up to you to define what constitutes a component for testing—it could be a method in a class or a set of methods that accomplish an essential purpose. For example, it could be an arithmetic operation, as in the calculator app used as an example in the Quick Start chapter. It could be the different methods that handle the interaction between the contents of a `UITableView` and a list of names you maintain in your code's data structures. Each one of those methods and operations implies a component of the app's functionality and a test to check it. The behavior of a component for testing should be completely deterministic; the test either passes or fails.

The more you can divide up the behavior of your app into components, the more effectively you can test that the behavior of your code meets the reference standards in all particulars as your project grows and changes. For a large project with many components, you'll need to run a large number of tests to test the project thoroughly. Tests should be designed to run quickly, when possible, but some tests are necessarily large and execute more slowly. Small, fast running tests can be run often and used when there is a failure in order to help diagnose and fix problems easily.

Tests designed for the components of a project are the basis of test-driven development, which is a style of writing code in which you write test logic before writing the code to be tested. This development approach lets you codify requirements and edge cases for your code before you implement it. After writing the tests, you develop your algorithms with the aim of passing the tests. After your code passes the tests, you have a foundation upon which you can make improvements to your code, with confidence that any changes to the expected behavior (which would result in bugs in your product) are identified the next time you run the tests.

Even when you're not using test-driven development, tests can help reduce the introduction of bugs in your code as you modify it to enhance features and functionality. You incorporate testing in a working app to ensure that future changes don't modify the app's existing behavior other than in your planned ways. As you fix bugs, you add tests that confirm that the bugs are fixed. Tests should exercise your code, looking for both expected successes and expected failures, to cover all the boundary conditions.

> **Note:** Adding tests to a project that was not designed with testing in mind may require redesigning or refactoring parts of the code to make it easier to test them. Appendix A: Writing Testable Code contains simple guidelines for writing testable code that you might find useful.

Components can encompass the interactions between any of the various parts of your app. Because some types of tests take much longer to run, you might want to run them only periodically or only on a server. As you'll see in the next chapters, you can organize your tests and run them in many different ways to meet different needs.

Performance Testing

Tests of components can be either functional in nature or measure performance. XCTest provides API to measure time-based performance, enabling you to track performance improvements and regressions in a similar way to functional compliance and regressions.

To provide a success or failure result when measuring performance, a test must have a *baseline* to evaluate against. A baseline is a combination of the average time performance in ten runs of the test method with a measure of the standard deviation of each run. Tests that drop below the time baseline or that vary too much from run to run are reported as failures.

> **Note:** The first time you run a performance measurement test, XCTest always reports failure since the baseline is unknown. Once you have accepted a certain measurement as a baseline, XCTest evaluates and reports success or failure, and provides you with a means to see the results of the test in detail.

User Interface Testing

The functional and performance testing discussed so far is generally referred to as unit testing, where a “unit” is the component of functionality that you have decided upon with respect to granularity and level. Unit testing is primarily concerned with forming good components that behave as expected and interact with other components as expected. From the design perspective, unit testing approaches your development project from its inside, vetting that the components fulfill your intent.

Users interact with these internals of your code through the user interface. User interface interactions are generally coarser-grained, higher level behaviors, taking external activity and integrating the operation of several components (subsystems) to call your app’s functions. It is difficult to write unit tests to exercise the interactions that users experience through the user interface without special facilities designed to operate the UI from outside the app context. Tests written with these special facilities are called “UI tests.”

UI tests approach testing apps from the external surface, as the users experience it. They enable you to write tests that send simulated events to both system-supplied and custom UI objects, capture the response of those objects, and then test that response for correctness or performance much like you do with the internally-oriented unit tests.

App and Library Tests

Xcode offers two types of unit test contexts: app tests and library tests.

- **App tests.** App tests check the correct behavior of code in your app, such as the example of the calculator app’s arithmetic operations.
- **Library tests.** Library tests check the correct behavior of code in dynamic libraries and frameworks independent of their use in an app’s runtime. With library tests you construct unit tests that exercise the components of a library.

Testing your projects using these test contexts as appropriate helps you maintain expected and anticipated behavior as your code evolves over time.

XCTest—the Xcode Testing Framework

XCTest is the testing framework supplied for your use, starting with Xcode 5.

Regarding versions and compatibility:

- In Xcode 5, XCTest is compatible with running on OS X v10.8 and OS X v10.9, and with iOS 7 and later.
- In Xcode 6, XCTest is compatible with running on OS X v10.9 and OS X v10.10, and with iOS 6 and later.
- In Xcode 7, XCTest is compatible with running on OS X v10.10 and OS X v10.11, and with iOS 6 and later.

  UI tests are supported running on OS X v10.11 and iOS 9, both in Simulator and on devices.

For more detailed version compatibility information, see *Xcode Release Notes*.

Xcode incorporates `XCTest.framework` into your project. This framework provides the APIs that let you design tests and run them on your code. For detailed information about the XCTest framework, see the XCTest Framework Reference.

> **Note:** For information about migrating OCUnit to XCTest, see Appendix B: Transitioning from OCUnit to XCTest.

Where to Start When Testing

When you start to create tests, keep the following ideas in mind:

- When creating unit tests, focus on testing the most basic foundations of your code, the Model classes and methods, which interact with the Controller.

  A high-level block diagram of your app most likely has Model, View, and Controller classes—it is a familiar design pattern to anyone who has been working with Cocoa and Cocoa Touch. As you write tests to cover all of your Model classes, you'll have the certainty of knowing that the base of your app is well tested before you work your way up to writing tests for the Controller classes—which start to touch other more complex parts of your app, for example, a connection to the network with a database connected behind a web service.

  As an alternative starting point, if you are authoring a framework or library, you may want to start with the surface of your API. From there, you can work your way in to the internal classes.

- When creating UI tests, start by considering the most common workflows. Think of what the user does when getting started using the app and what UI is exercised immediately in that process. Using the UI recording feature is a great way to capture a sequence of user actions into a UI test method that can be expanded upon to implement tests for correctness and/or performance.

  UI tests of this type tend to start with a relatively coarse-grained focus and might cut across several subsystems; they can return a lot of information that can be hard to analyze at first. As you work with your UI test suite, you can refine the testing granularity and focus UI tests to reflect specific subsystem behaviors more clearly.