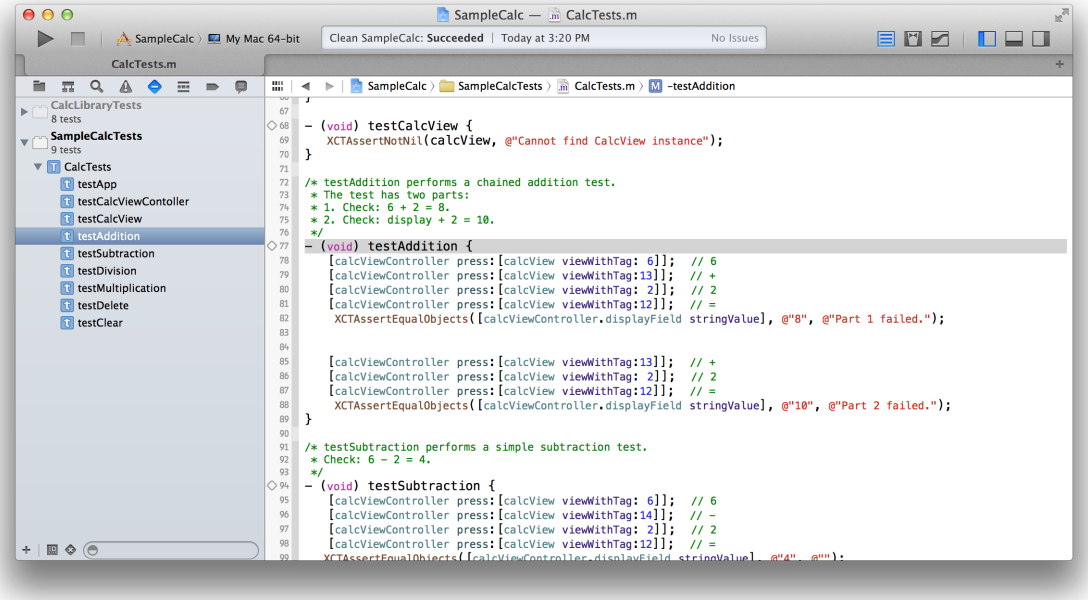
Testing with Xcode

Developer

Contents

About Testing with Xcode

Xcode provides you with capabilities for extensive software testing. Testing your projects enhances robustness, reduces bugs, and speeds the acceptance of your products for distribution and sale. Well-tested apps that perform as expected improve user satisfaction. Testing can also help you develop your apps faster and further, with less wasted effort, and can be used to help multiperson development efforts stay coordinated.



At a Glance

In this document you'll learn how to use the testing features included in Xcode. XCTest is the testing framework new in Xcode 5, replacing OCUnit. This framework is automatically linked by all new test targets.

- **Quick Start.** Xcode 5 and later has streamlined and automated the process of configuring projects for testing with the test navigator to ease bringing tests up and running.
- **Performance Measurement.** Xcode 6 and later includes the ability to create tests that allow you to measure and track performance changes against a baseline.
- **Xcode Server and Continuous Integration.** Xcode tests can be configured using bots to execute on a collection of devices connected to a Mac running OS X Server automatically.

- **Modernization.** Xcode includes a migrator for converting projects that have OCUnit tests to have XCTest tests.

Prerequisites

You should be familiar with app design and programming concepts.

See Also

See these session videos from WWDC for a good look at Xcode testing capabilities.

- WWDC 2013: Testing in Xcode 5 (409)
- WWDC 2014: Testing in Xcode 6 (414)

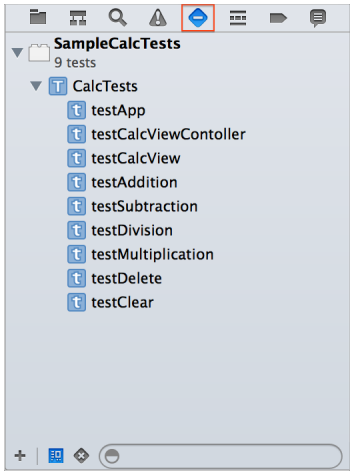
Quick Start

The intent of this quick start is to show that you can make testing an integral part of your software development, and that testing is convenient and easy to work with.

Introducing the Test Navigator

You'll use the Xcode test navigator often when you are working with tests.

The test navigator is the part of the workspace designed to ease your ability to create, manage, run, and review tests. You access it by clicking its icon in the navigator selector bar, located between the issue navigator and the debug navigator. When you have a project with a suite of tests defined, you see a navigator view similar to the one shown here.



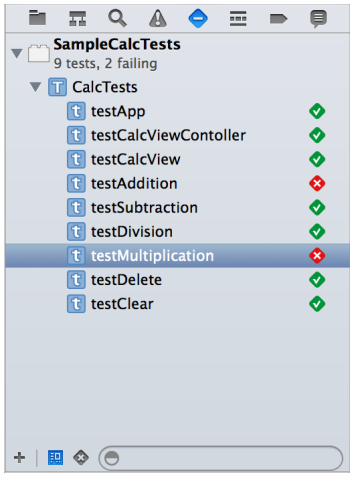
The test navigator shown above displays a hierarchical list of the test bundles, classes, and methods included in a sample project. This particular project is a sample calculator app. The calculator engine is implemented as a framework. You can see at the top level of the hierarchy the `SampleCalcTests` test bundle, for testing the code in the application.

Note: Xcode test targets produce test bundles that are displayed in the test navigator.

If your tests use assets—data files, images, and so forth—they can be added to the test bundle and accessed at run time using the `NSBundle` APIs. Using `+[NSBundle bundleForClass:]` with your test class ensures that you obtain the correct bundle to retrieve assets. For more information, see *NSBundle Class Reference*.

Xcode schemes control what is built. Schemes also control which of the available test methods to execute for the test action. You can enable and disable test bundles, classes, and methods selectively by Control-clicking the items in the test navigator list and choosing Enable or Disable from the shortcut menu, thereby enabling or disabling the items in the scheme.

The active test bundle in this view is `SampleCalcTests`. `SampleCalcTests` includes one test class, which in turn contains nine test methods. The Run button ( ) appears to the right of the item name when you hold the pointer over any item in the list. This is a quick way to run all the tests in a bundle, all the tests in the class, or any individual test. Tests return pass or fail results to Xcode. As tests are being executed, these indicators update to show you the results, a green checkmark for pass or a red x for fail. In the test navigator shown here, two of the tests have asserted a failure.



Clicking any test class or test method in the list opens the test class in the source editor. Test classes and test methods are marked in the source editor gutter with indicators as well, which work in the same way that they do in the test navigator. Test failures display the result string at the associated assertions in the source editor.

At the bottom of the test navigator is the Add button (+) as well as filtering controls. You can narrow the view to just tests in the active scheme or just failed tests, and you can also filter by name.

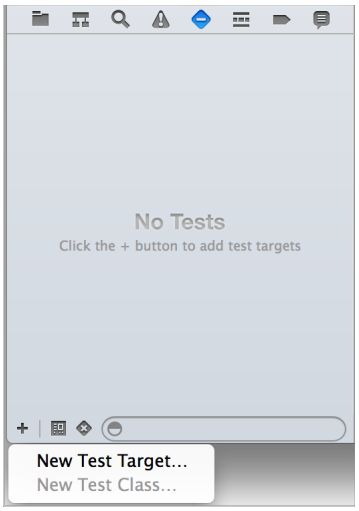
For more detailed information on test navigator operations, see *Test Navigator Help*.

Add Testing to Your App

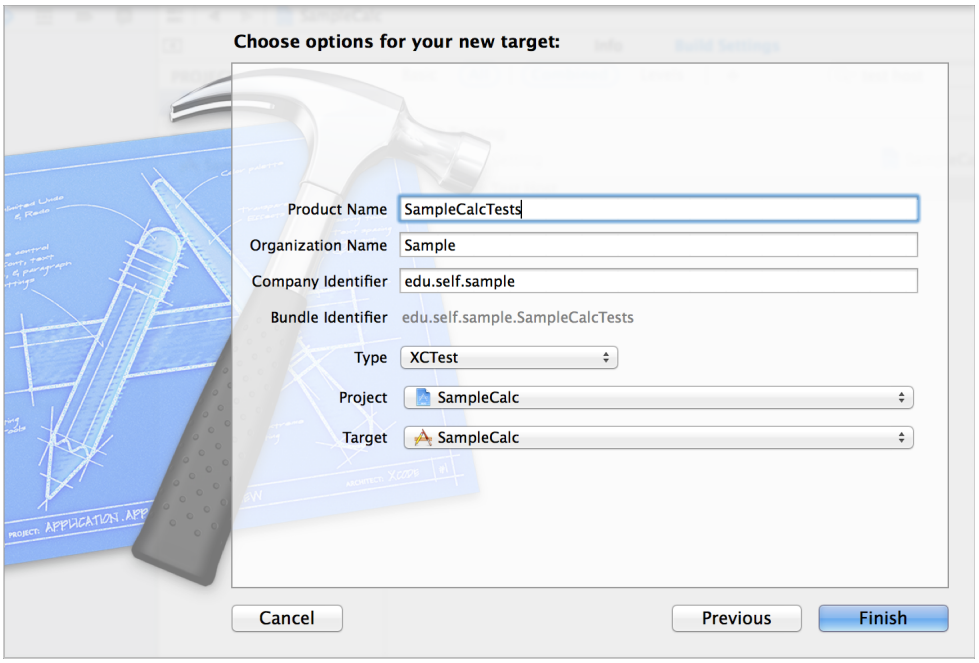
New app, framework, and library projects created in Xcode 5 or later are preconfigured with a test target. When you start with a new project and open the test navigator, you see a test bundle, a test class, and a template test method. But you might be opening a preexisting project from an earlier version of Xcode that has no test targets defined yet. The workflow presented here assumes a preexisting project with no tests incorporated.

Create a Test Target

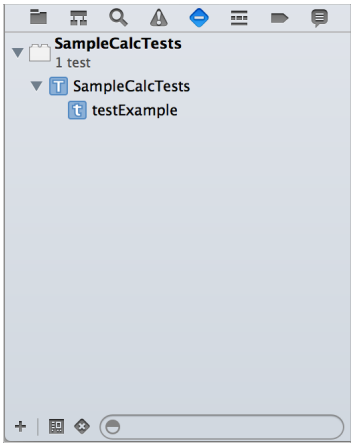
With the test navigator open, click the Add button (+) in the bottom-left corner and choose New Test Target from the menu.



In the new target assistance that appears, edit the Product Name and other parameters to your preferences and needs.

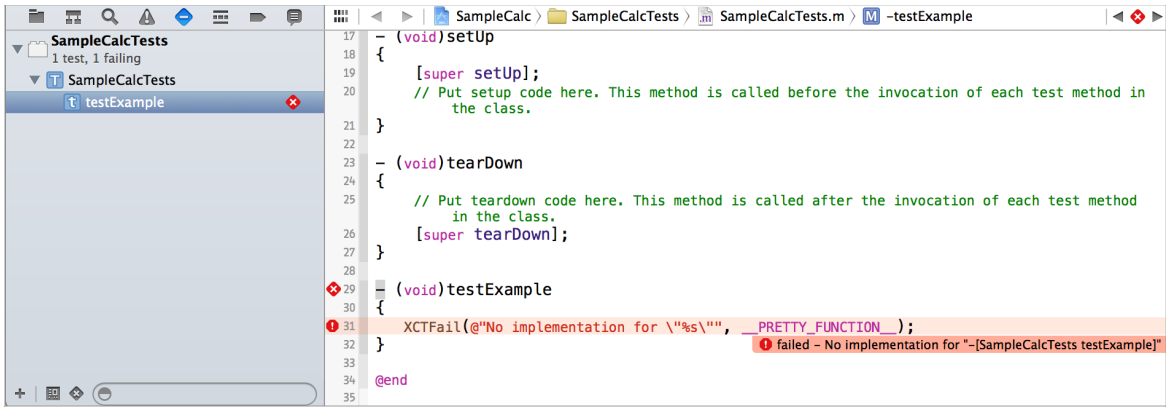


Click the Finish button to add your target, which contains a template test class and a test method to the test navigator view.



Run the Test and See the Results

Now that you've added testing to your project, you want to develop the tests to do something useful. But first, hold the pointer over the `SampleCalcTests` test class in the test navigator and click the Run button to run all the test methods in the class. A test failure is indicated in the test navigator. Click the `testExample` method to see the following view of the test results, source code, and highlighted error condition:



The test failed because the test class template contains a default test method that calls `XCTFail()`, an assertion that forced an unconditional failure message.

Edit the Test and Run It Again

Because this sample project is a calculator app, you might decide first to check whether it performs addition correctly. Because tests are built in the app project, you can add all the context and other information needed to perform tests at whatever level of complexity makes sense for your needs.

Insert the following #import and instance variable declarations into the `SampleCalcTests.m` file:

```
#import <XCTest/XCTest.h>
//
// Import the application specific header files
#import "CalcViewController.h"
#import "CalcAppDelegate.h"

@interface CalcTests : XCTestCase {
// add instance variables to the CalcTests class
@private
    NSApplication       *app;
    CalcAppDelegate     *appDelegate;
```

```
    CalcViewController  *calcViewController;
    NSView              *calcView;
}
@end
```

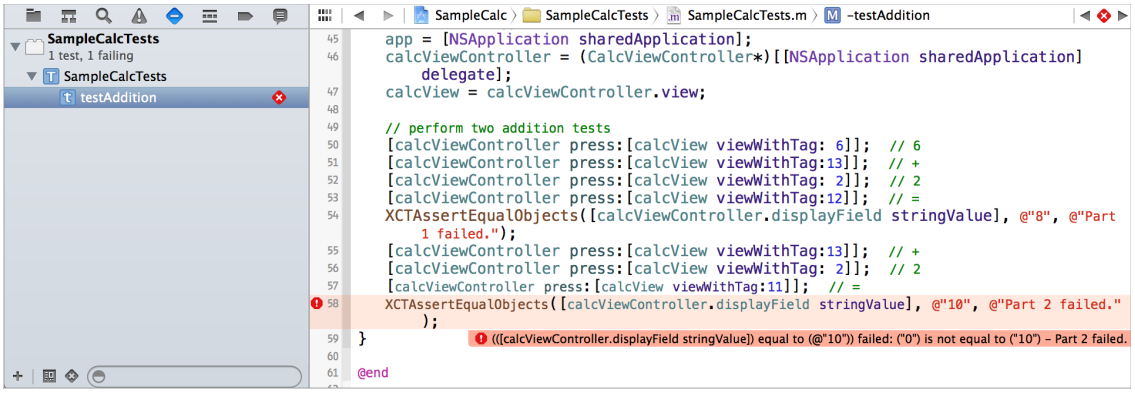
Then give the test method a descriptive name, such as `testAddition`, and add the implementation source for the method.

```
- (void) testAddition
{
   // obtain the app variables for test access
   app                  = [NSApplication sharedApplication];
   calcViewController   = (CalcViewController*)[[NSApplication sharedApplication]
 delegate];
   calcView             = calcViewController.view;


   // perform two addition tests
   [calcViewController press:[calcView viewWithTag: 6]];  // 6
   [calcViewController press:[calcView viewWithTag:13]];  // +
   [calcViewController press:[calcView viewWithTag: 2]];  // 2
   [calcViewController press:[calcView viewWithTag:12]];  // =
    XCTAssertEqualObjects([calcViewController.displayField stringValue], @"8",
@"Part 1 failed.");


   [calcViewController press:[calcView viewWithTag:13]];  // +
   [calcViewController press:[calcView viewWithTag: 2]];  // 2
   [calcViewController press:[calcView viewWithTag:12]];  // =
    XCTAssertEqualObjects([calcViewController.displayField stringValue], @"10",
@"Part 2 failed.");
}
```

Now use the Run button in the test navigator (or the indicator in the source editor) to run the `testAddition` method.



Notice that the list in the test navigator changed to reflect that `testExample` has been replaced by `testAddition`.

There's still an error here. Looking at the source, Part 1 succeeded—it is Part 2 that has a problem. On closer examination, the error is obvious: The reference string "`11`" is off by 1—a typo. Changing the reference string to "`10`" fixes the problem and the test succeeds.

Use the setUp() and tearDown() Methods for Common Code

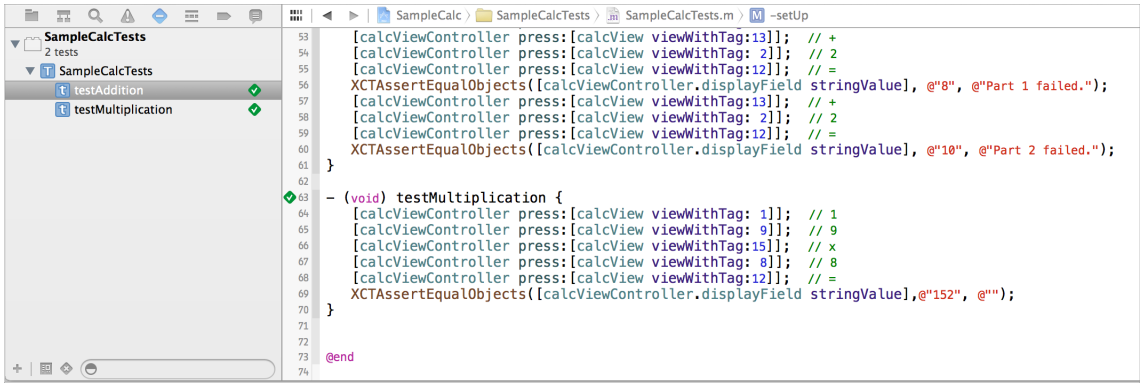
Xcode runs test methods one at a time for all the test classes in the active test bundle. In this small example only the one test method was implemented in the test class, and it needed access to three of the calculator app variable objects to function. If you wrote four or five test methods in this same class, you might find yourself repeating the same code in every test method to obtain access to the app object state. The XCTest framework provides you with instance methods for test classes, `setUp` and `tearDown`, which you can use to put such common code called before and after runs each test method runs.

Using `setUp` and `tearDown` is simple. From the `testAddition` source in `Mac_Calc_Tests.m`, cut the four lines starting with `// obtain the app variable for test access` and paste them into the default `setUp` instance method provided by the template.

```
- (void)setUp
{
    [super setUp];
    // Put setup code here. This method is called before the invocation of each
test method in the class.

   // obtain the app variables for test access
   app                  = [NSApplication sharedApplication];
   calcViewController   = (CalcViewController*)[[NSApplication sharedApplication]
 delegate];
   calcView             = calcViewController.view;
}
```

Now add a second test method, `testMultiplication`, and others, with minimal duplicated code.



Summary

As you can see from this short Quick Start, it is simple to add testing to a project. Here are some things to notice:

- Xcode sets up most of the basic testing configuration. When you add a test target, Xcode creates the test bundle files for the project, adds the test bundle to the test navigator, adds the XCTest framework to the project, and provides templates for test classes and test methods.

- The test navigator lets you locate and edit test methods easily. You can run tests immediately using the indicator buttons in the test navigator or directly from the source editor when a test class implementation is open. When a test fails, indicators in the test navigator are paired with failure markers in the source editor.
- A single test method can include multiple assertions, resulting in a single pass or fail result. This approach enables you to create simple or very complex tests depending on your project's needs.
- The `setup` and `tearDown` instance methods provide you with a way to factor common code used in many test methods for greater consistency and easier debugging.

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. Unit testing typically deals with testing smaller components at the foundation levels of the project. XCTest allows you to write tests for components at any level of the hierarchy.

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 (page 7) chapter. It could be the different methods that handle the interaction between the contents of a `TableView` 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 behavior. 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. Writing Testable Code (page 55) 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. The Xcode 6 version of XCTest has been enhanced with 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.

App and Library Tests

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

- **App tests.** App tests check the components of code in your app, such as the example of the calculator app's arithmetic operations. You can use app tests to ensure that the connections of your user-interface controls (outlets and actions) remain in place and that your controls and controller objects work correctly with your object model as you work on your app.
- **Library tests.** Library tests check the correct behavior of standalone code (code not running in an app). With library tests you put together tests to exercise the components of a library, usually the library's objects and methods. You can also use library tests to perform stress-testing of your code to ensure that it behaves correctly in extreme situations that are unlikely in a running app. These tests help you produce robust code that works correctly when used in ways that you did not anticipate.

XCTest—the Xcode Testing Framework

XCTest is the testing framework available in Xcode 5 and later.

> **Note on versions and compatibility:** In Xcode 5.x, XCTest is compatible with running on OS X v10.8 and OS X v10.9, and with iOS 7.x and later. In Xcode 6.x, XCTest is compatible with running on OS X v10.9 and OS X v10.10, and with iOS 6.x and later. For more detailed version compatibility information, see *Xcode Release Notes*.

If you have used OCUnit testing with Xcode before, you may recognize similarities between XCTest and OCUnit. XCTest is a modernized reimplementation of OCUnit that offers better integration with Xcode and lays the foundation for future improvements to Xcode testing capabilities. Instead of `SenTestingKit.framework`, Xcode incorporates `XCTest.framework` into your project. This framework provides APIs that let you design tests and run them on your code.

Note: Xcode includes a migrator for updating projects that have existing OCUnit tests. For more information about migrating OCUnit to XCTest, see Transitioning from OCUnit to XCTest (page 59).

Where to Start When Testing

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

- 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 could work your way in to the internal classes.

In the rest of this document you learn how to create, run, and debug tests for your development project using the tools provided by Xcode.

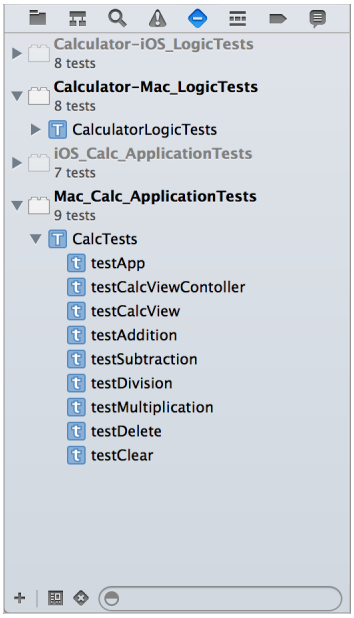
Writing Test Classes and Methods

When you add a test target to a project with the test navigator, Xcode displays the test classes and methods from that target in the test navigator. In the test target are the test classes containing test methods. This chapter explains how you create test classes and write test methods.

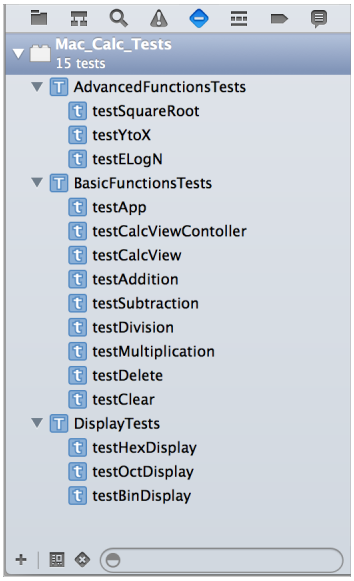
Test Targets, Test Bundles, and the Test Navigator

Before looking at creating test classes, it is worth taking another look at the test navigator. Using it is central to creating and working with tests.

The test navigator lays out the components of all test bundles in the project, displaying the test classes and test methods in a hierarchical list. Here's a test navigator view for a project that has multiple test targets, showing the nested hierarchy of test bundles, test classes, and test methods.



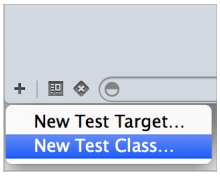
Test bundles can contain multiple test classes. You can use test classes to segregate tests into related groups, either for functional or organizational purposes. For example, for the calculator example project you might create `BasicFunctionsTests`, `AdvancedFunctionsTests`, and `DisplayTests` classes, all part of the `Mac_Calc_Tests` test bundle.



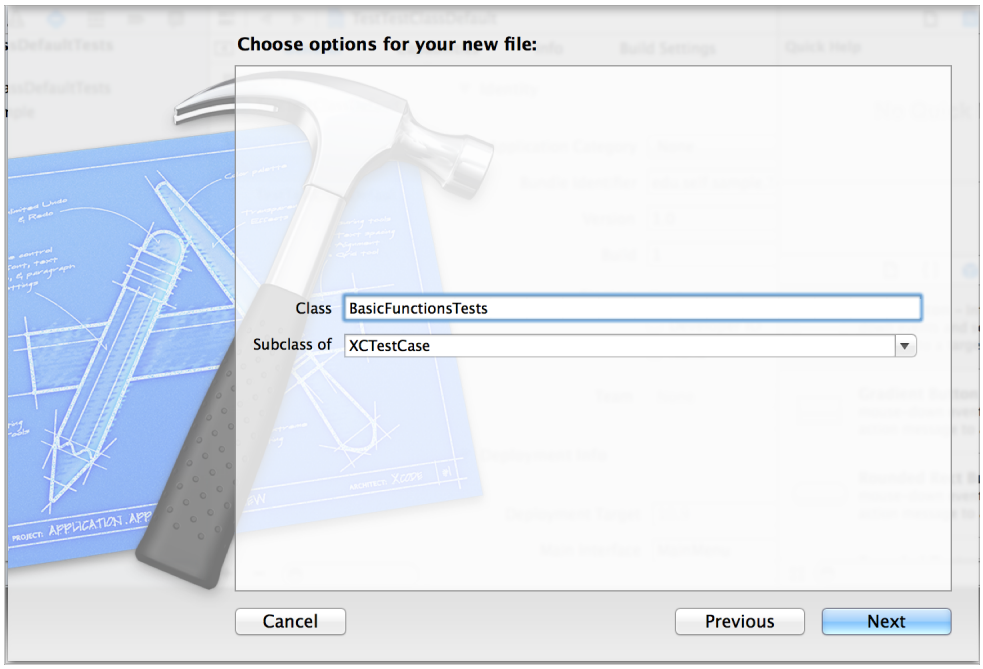
Some types of tests might share certain types of setup and teardown requirements, making it sensible to gather those tests together into classes, where a single set of setup and teardown methods can minimize how much code you have to write for each test method.

Creating a Test Class

You use the Add button (+) and the New Test Class command in the test navigator to create new test classes.

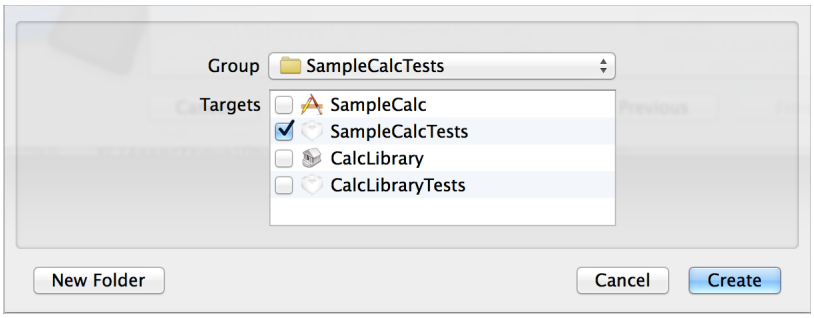


Each test class you add results in a file named *TestClassName* .m being added to the project, based on the test class name you enter in the configuration sheet.



Note: All test classes are subclasses of `XCTestCase`, provided by the XCTest framework.

Although by default Xcode organizes test class implementation files into the group it created for your project's test targets, you can organize the files in your project however you choose. The standard Xcode Add Files sheet follows this configuration when you press the Next button.



You use the Add Files sheet the same way as when adding new files to the project in the project navigator. For details on how to use the Add Files sheet, see Adding an Existing File or Folder.

Note: When you create a new project, a test target and associated test bundle are created for you by default with names derived from the name of your project. For instance, creating a new project named `MyApp` automatically generates a test bundle named `MyAppTests` and a test class named `MyAppTests` with the associated `MyAppTests.m` implementation file.

Test Class Structure

Test classes have this basic structure:

```
#import <XCTest/XCTest.h>

@interface MyAppTests : XCTestCase
@end

@implementation MyAppTests

// setUp and tearDown
- (void)setUp
{
    [super setUp];
    // Put additional setup code here.
}

- (void)tearDown
{
    // Put additional teardown code here.
    [super tearDown];
}

// test methods
- (void)testXXXX
{
    // setup code
    // test logic and XCTest assertions go here.
    // teardown code
```

```
}
@end
```

The test class is implemented in Objective-C in this example, but can also be implemented in Swift.

> **Note:** The implementation examples in this text are all written in Objective-C for consistency.
>
> Swift is fully compatible with using XCTest and implementing your test methods. All Swift and Objective-C cross-language implementation capabilities can be used as well.

Notice that the implementation contains methods for instance setup and teardown with a basic implementation; these methods are required. If all of the test methods in a class require the same code, you can customize `setUp` and `tearDown` to include it. The code you add runs before and after each test method runs. You can optionally add customized methods for class setup (`+ (void)setUp`) and teardown (`+ (void)tearDown`) as well, which run before and after all of the test methods in the class.

Xcode executes tests to make the use of these methods clear, as discussed in the following sections.

Flow of Test Execution

During test execution, XCTest finds all the test classes and, for each class, runs all of its test methods. (All test classes inherit from `XCTestCase`.)

For each class, testing starts by running the class setup method. For each test method, a new instance of the class is allocated and its instance setup method executed. After that it runs the test method, and after that the instance teardown method. This sequence repeats for all the test methods in the class. After the last test method teardown in the class has been run, Xcode executes the class teardown method and moves on to the next class. This sequence repeats until all the test methods in all test classes have been run.

Writing Test Methods

You add tests to a test class by writing test methods. A test method is an instance method of a test class that begins with the prefix *test*, takes no parameters, and returns `void`, for example, `(void) testColorIsRed()`. A test method calls code in your project and, if that code does not produce the expected result, reports failures using a set of assertion APIs. For example, a function's return value might be compared against an expected value or your test might assert that improper use of a method in one of your classes throws an exception. XCTest Assertions (page 28) describes these assertions.

For a test method to access the code to be tested, import the corresponding header files into your test class.

When Xcode runs tests, it invokes each test method independently. Therefore, each method must prepare and clean up any auxiliary variables, structures, and objects it needs to interact with the subject API. If this code is common to all test methods in the class, you can add it to the required `setUp` and `tearDown` instance methods described in Test Class Structure (page 23).

Here is the model of a test method:

```
- (void)testColorIsRed {
   ...    // Set up, call test subject API. (Code could be shared in setUp method.)
   ...     // Test logic and values, assertions report pass/fail to testing
framework.
   ...     // Tear down. (Code could be shared in tearDown method.
}
```

And here is a simple test method example that checks to see whether the `CalcView` instance was successfully created for SampleCalc, the app shown in the Quick Start (page 7) chapter:

```
- (void) testCalcView {
   // setup
   app = [NSApplication sharedApplication];
   calcViewController = (CalcViewController*)[NSApplication sharedApplication]
delegate];
   calcView             = calcViewController.view;

   XCTAssertNotNil(calcView, @"Cannot find CalcView instance");
   // no teardown needed
}
```

Writing Tests of Asynchronous Operations

Tests execute synchronously because each test is invoked independently one to the next. But more and more system functions execute asychronously. To handle testing components which call asynchronously executing methods and functions, XCTest has been enhanced in Xcode 6 to include the ability to handle blocks using new API and objects of class `XCTestExpectation`. These objects respond to new XCTest methods that allow the test method to wait until either the async call returns or a timeout is reached.

A source example:

```
// Test that the document is opened. Because opening is asynchronous,
// use XCTestCase's asynchronous APIs to wait until the document has
// finished opening.

- (void)testDocumentOpening
{
    // Create an expectation object.
    // This test only has one, but it's possible to wait on multiple expectations.
    XCTestExpectation *documentOpenExpectation = [self
expectationWithDescription:@"document open"];

    NSURL *URL = [[NSBundle bundleForClass:[self class]]
                          URLForResource:@"TestDocument" withExtension:@"mydoc"];
    UIDocument *doc = [[UIDocument alloc] initWithFileURL:URL];
    [doc openWithCompletionHandler:^(BOOL success) {
        XCTAssert(success);
        // Possibly assert other things here about the document after it has
opened...

        // Fulfill the expectation—this will cause -waitForExpectation
        // to invoke its completion handler and then return.
        [documentOpenExpectation fulfill];
    }];

    // The test will pause here, running the run loop, until the timeout is hit
    // or all expectations are fulfilled.
    [self waitForExpectationsWithTimeout:1 handler:^(NSError *error) {
        [doc closeWithCompletionHandler:nil];
    }];
}
```

For more details on writing methods for asynchronous operations, see the `XCTestCase+AsynchronousTesting.h` header file in `XCTest.framework`.0

Writing Performance Tests

A performance test takes a block of code that you want to evaluate and runs it ten times, collecting the average execution time and the standard deviation for the runs. These statistics combine to create a baseline for comparison, a means to evaluate success or failure. To implement performance measuring tests, you write methods using new API from XCTest in Xcode 6 and later.

```
- (void)testPerformanceExample {
    // This is an example of a performance test case.
    [self measureBlock:^{
        // Put the code you want to measure the time of here.
    }];
}
```

The following simple example shows a performance test written to test addition speed with the calculator sample app. A `measureBlock:` is added along with an iteration for XCTest to time.

```
- (void) testAdditionPerformance {
    [self measureBlock:^{
        // set the initial state
        [calcViewController press:[calcView viewWithTag: 6]];  // 6
        // iterate for 100000 cycles of adding 2
        for (int i=0; i<100000; i++) {
        [calcViewController press:[calcView viewWithTag:13]];  // +
        [calcViewController press:[calcView viewWithTag: 2]];  // 2
        [calcViewController press:[calcView viewWithTag:12]];  // =
        }

    }];
}
```

Performance tests, once run, provide information in the source editor when viewing the implementation file, in the issues navigator, and in the reports navigator. Clicking on the information presents individual run values. The results display includes controls to set the resulsts as the baseline for future runs of the tests. Baselines are stored per-device-configuration, so you can have the same test executing on several different devices and have each maintain a different baseline dependent upon the specific configuration's processor speed, memory, and so forth.

> **Note:** Performance measuring tests always report failure on the first run and until a baseline value is set on a particular device configuration.

For more details on writing methods for performance measuring tests, see the `XCTestCase.h` header file in `XCTest.framework`.

XCTest Assertions

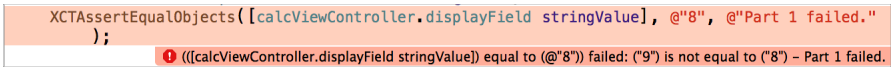
Your test methods use assertions provided by the XCTest framework to present test results that Xcode displays. All assertions have a similar form: items to compare or a logical expression, a failure result string format, and the parameters to insert into the string format.

> **Note:** The last parameter of all assertions is `format...`, a format string and its variable parameter list. XCTest provides a default failure result string for all assertions, assembled using the parameters passed to the assertion. The `format` string offers the ability to provide an optional additional custom description of the failure that you can choose to supply in addition to the provided description. This parameter is optional and can be completely omitted.

For example, look at this assertion in the `testAddition` method presented in Quick Start (page 7):

```
XCTAssertEqualObjects([calcViewController.displayField stringValue], @"10", @"Part 2 failed.");
```

Reading this as plain language, it says *"Indicate a failure when a string created from the value of the controller's display field is not the same as the reference string '8'."* Xcode signals with a fail indicator in the test navigator if the assertion fails, and Xcode also displays a failure with the description string in the issues navigator, source editor, and other places. A typical result in the source editor looks like this:



Test methods can include multiple assertions. Xcode signals a test method failure if any of the assertions it contains reports a failure.

Assertions fall into five categories: unconditional fail, equality tests, nil tests, Boolean tests, and exception tests.

Using Assertions with Objective-C and Swift

When using XCTest assertions, know the fundamental differences in the assertions' compatibility and behavior when writing Objective-C (and other C-based languages) code and when writing Swift code. Understanding these differences makes writing and debugging your tests easier.

XCTest assertions that perform equality tests are divided between those that compare objects and those that compare nonobjects. For example, `XCTAssertEqualObjects` tests equality between two expressions that resolve to an object type. And `XCTAssertEqual` tests equality between two expressions that resolve to the value of a scalar type. This difference is marked in the XCTest assertions listing by including "this test is for scalars" in the description. Marking assertions with "scalar" this way informs you of the basic distinction, but it is not an exact description of which expression types are compatible.

- For Objective-C, assertions marked for scalar types can be used with the types that can be used with the equality comparison operators: ==, `!=`, <=, <, >=, and >. If the expression resolves to any C type, struct, or array comparison that works with these operators, it is considered a scalar.
- For Swift, assertions marked for scalars can be used to compare any expression type that conforms to the `Equatable` protocol (for all of the "equal" and "not equal" assertions) and `Comparable` protocol (for the "greater than" and "less than" assertions). In addition, assertions marked for scalars have overrides for `[T]` and for `[K:V]`, where `T`, `K`, and `V` conform to `Equatable` or `Comparable` protocols. For example, arrays of an equatable type are compatible with `XCTAssertEqual`, and dictionaries whose keys and values are both comparable types are compatible with `XCTAssertLessThan`.

Using XCTest assertions in your tests also differs between Objective-C and Swift because of how the languages differ in treating data types and implicit conversions.

- For Objective-C, the use of implicit conversions in the XCTest implementation allows the comparisons to operate independent of the expressions' data types, and no check is made of the input data types.
- For Swift, implicit conversions are not allowed because of the relative lack of safety inherent to implicit type conversions, so both parameters to a comparison must be of the same type. Type mismatches are flagged at compile time and in the source editor.

Assertions Listed by Category

The following sections list the XCTest assertions. You can obtain more information on XCTest assertions by referencing `XCTestAssertions.h` in Xcode using Quick Help.

Unconditional Fail

XCTFail. Generates a failure unconditionally

```
XCTFail(format...)
```

Equality Tests

XCTAssertEqualObjects. Generates a failure when `expression1` is not equal to `expression2` (or one object is nil and the other is not).

```
XCTAssertEqualObjects(expression1, expression2, format...)
```

XCTAssertNotEqualObjects. Generates a failure when `expression1` is equal to `expression2`.

```
XCTAssertNotEqualObjects(expression1, expression2, format...)
```

XCTAssertEqual. Generates a failure when `expression1` is not equal to `expression2`. This test is for scalars.

```
XCTAssertEqual(expression1, expression2, format...)
```

XCTAssertNotEqual. Generates a failure when `expression1` is equal to `expression2`. This test is for scalars.

```
XCTAssertNotEqual(expression1, expression2, format...)
```

XCTAssertEqualWithAccuracy. Generates a failure when the difference between `expression1` and `expression2` is greater than `accuracy`. This test is for scalars such as floats and doubles, where small differences could make these items not exactly equal, but works for all scalars.

```
XCTAssertEqualWithAccuracy(expression1, expression2, accuracy, format...)
```

XCTAssertNotEqualWithAccuracy. Generates a failure when the difference between `expression1` and `expression2` is less than or equal to `accuracy`. This test is for scalars such as floats and doubles, where small differences could make these items not exactly equal, but works for all scalars.

```
XCTAssertNotEqualWithAccuracy(expression1, expression2, accuracy, format...)
```

XCTAssertGreaterThan. Generates a failure when `expression1` is less than or equal to `expression2`. This test is for scalar values.

```
XCTAssertGreaterThan(expression1, expression2, format...)
```

XCTAssertGreaterThanOrEqual. Generates a failure when `expression1` is less than `expression2`. This test is for scalar values.

```
XCTAssertGreaterThanOrEqual(expression1, expression2, format...)
```

XCTAssertLessThan. Generates a failure when `expression1` is greater than or equal to `expression2`. This test is for scalar values.

```
XCTAssertLessThan(expression1, expression2, format...)
```

XCTAssertLessThanOrEqual. Generates a failure when `expression1` is greater than `expression2`. This test is for scalar values.

```
XCTAssertLessThanOrEqual(expression1, expression2, format...)
```

Nil Tests

XCTAssertNil. Generates a failure when the `expression` parameter is not `nil`.

```
XCTAssertNil(expression, format...)
```

XCTAssertNotNil. Generates a failure when the `expression` parameter is `nil`.

```
XCTAssertNotNil(expression, format...)
```

Boolean Tests

XCTAssertTrue. Generates a failure when `expression` evaluates to `false`.

```
XCTAssertTrue(expression, format...)
```

XCTAssert. Generates a failure when `expression` evaluates to `false`. Synonymous with `XCTAssertTrue`.

```
XCTAssert(expression, format...)
```

XCTAssertFalse. Generates a failure when `expression` evaluates to `true`.

```
XCTAssertFalse(expression, format...)
```

Exception Tests

XCTAssertThrows. Generates a failure when `expression` does not throw an exception.

```
XCTAssertThrows(expression, format...)
```

XCTAssertThrowsSpecific. Generates a failure when `expression` does not throw an exception of a specific class.

```
XCTAssertThrowsSpecific(expression, exception_class, format...)
```

XCTAssertThrowsSpecificNamed. Generates a failure when `expression` does not throw an exception of a specific class with a specific name. Useful for those frameworks like AppKit or Foundation that throw generic `NSException` with specific names (`NSInvalidArgumentException` and so forth).

```
XCTAssertThrowsSpecificNamed(expression, exception_class, exception_name, format...)
```

XCTAssertNoThrow. Generates a failure when an `expression` does throw an exception.

```
XCTAssertNoThrow(expression, format...)
```

XCTAssertNoThrowSpecific. Generates a failure when `expression` does throw an exception of the specified class. Any other exception is OK; that is, it does not generate a failure.

```
XCTAssertNoThrowSpecific(expression, exception_class, format...)
```

XCTAssertNoThrowSpecificNamed. Generates a failure when `expression` does throw an exception of a specific class with a specific name. Useful for those frameworks like AppKit or Foundation that throw generic NSException with specific names (`NSInvalidArgumentException` and so forth).

```
XCTAssertNoThrowSpecificNamed(expression, exception_class, exception_name, format...)
```

Running Tests and Viewing Results

Running tests and viewing the results is easy to do using the Xcode test navigator, as you saw in Quick Start (page 7). There are several additional interactive ways to run tests. Xcode runs tests based on what test targets are included and enabled in a scheme. The test navigator allows you to directly control which test targets, classes, and methods are included, enabled, or disabled in a scheme without having to use the scheme editor.

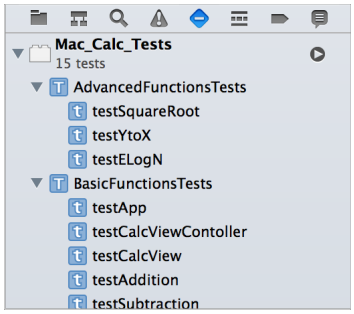
Commands for Running Tests

The test navigator provides you an easy way to run tests as part of your programming workflow. Tests can also be run either directly from the source editor or using the Product menu.

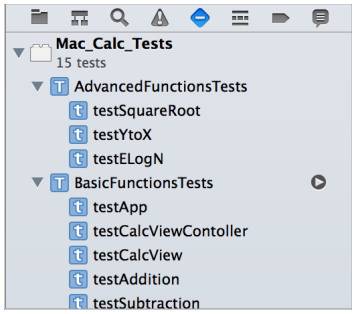
Using the Test Navigator

When you hold the pointer over a bundle, class, or method name in the test navigator, a Run button appears. You can run one test, all the tests in a class, or all the tests in a bundle depending on where you hold the pointer in the test navigator list.

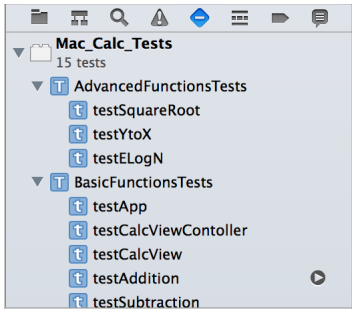
- To run all tests in a bundle, hold the pointer over the test bundle name and click the Run button that appears on the right.



- To run all tests in a class, hold the pointer over the class name and click the Run button that appears on the right.

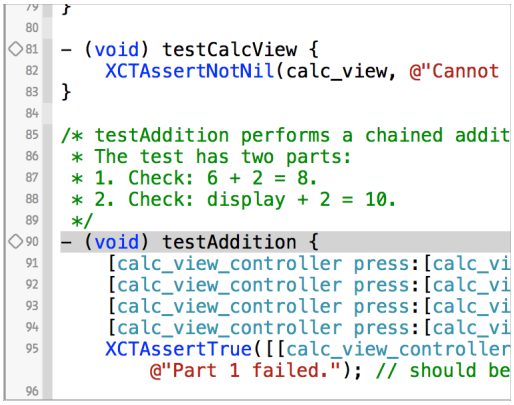


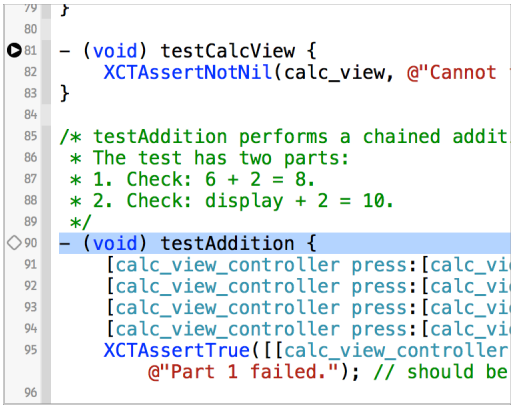
- To run a single test, hold the pointer over the test name and click the Run button that appears on the right.



Using the Source Editor

When you have a test class open in the source editor, a clear indicator appears in the gutter next to each test method name. Holding the pointer over the indicator displays a Run button. Clicking the Run button runs the test method, and the indicator then displays the pass or fail status of a test. Holding the pointer over the indicator will again display the Run button to repeat the test. This mechanism always runs just the one test at a time.





> **Note:** The same indicator appears next to the `@implementation` for the class as well, allowing you to run all of the tests in the class.

Using the Product Menu

The Product menu includes quickly accessible commands to run tests directly from the keyboard.

Product > Test. Runs the currently active scheme. The keyboard shortcut is Command-U.

Product > Build for > Testing and **Product > Perform Action > Test without Building.** These two commands can be used to build the test bundle products and run the tests independent of one another. These are convenience commands to shortcut the build and test processes. They're most useful when

changing code to check for warnings and errors in the build process, and for speeding up testing when you know the build is up to date. The keyboard shortcuts are Shift-Command-U and Control-Command-U, respectively.

Product > Perform Action > Test <testName>. This dynamic menu item senses the current test method in which the editing insertion point is positioned when you're editing a test method and allows you to run that test with a keyboard shortcut. The command's name adapts to show the test it will run, for instance, Product > Perform Action > Test *testAddition*. The keyboard shortcut is Control-Option-Command-U.

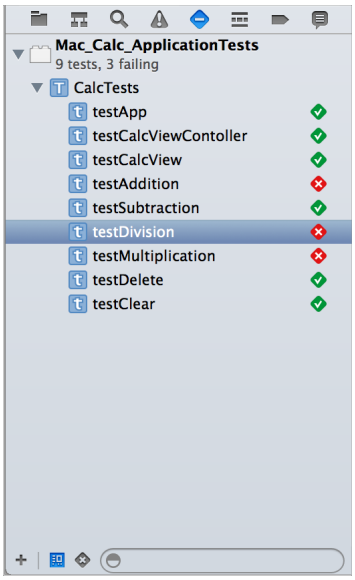
> **Note:** In addition to the source editor, this command also operates based on the selection in the project navigator and the test navigator. When either of those two navigators is active, the source editor does not have focus and the command takes the current selection in either of these navigators for input.
>
> In the test navigator, the selection can be on a test bundle, class, or method. In the project navigator, the selection can be on the test class implementation file, for instance, `CalcTests.m`.

Product > Perform Action > Test Again <testName>. Reruns the last test method executed, most useful when debugging/editing code in which a test method exposed a problem. Like the Product > Perform Action > Test command, the name of the test that is run appears in the command, for example, Product > Perform Action > Test Again *testAddition*. The keyboard shortcut is Control-Option-Command-G.

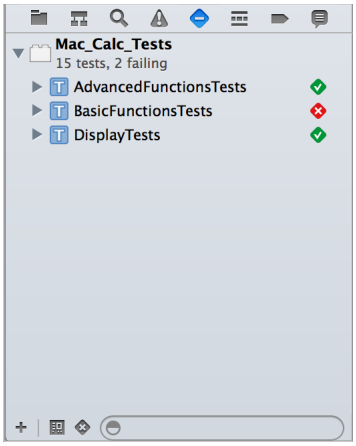
Display of Test Results

The XCTest framework displays the pass or fail results of a test method to Xcode in several ways. The following screenshots show where you can see the results.

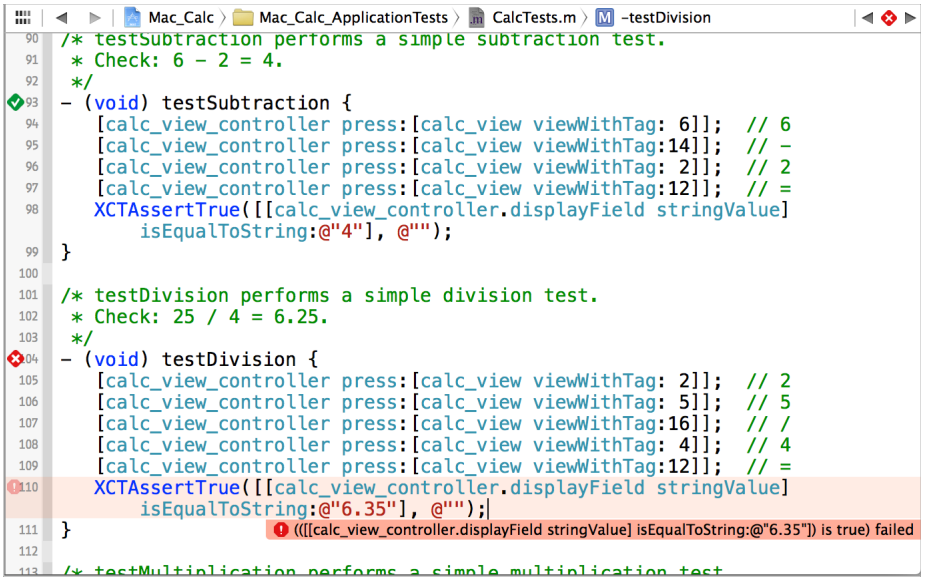
- In the test navigator, you can view pass/fail indicators after a test or group of tests is run.



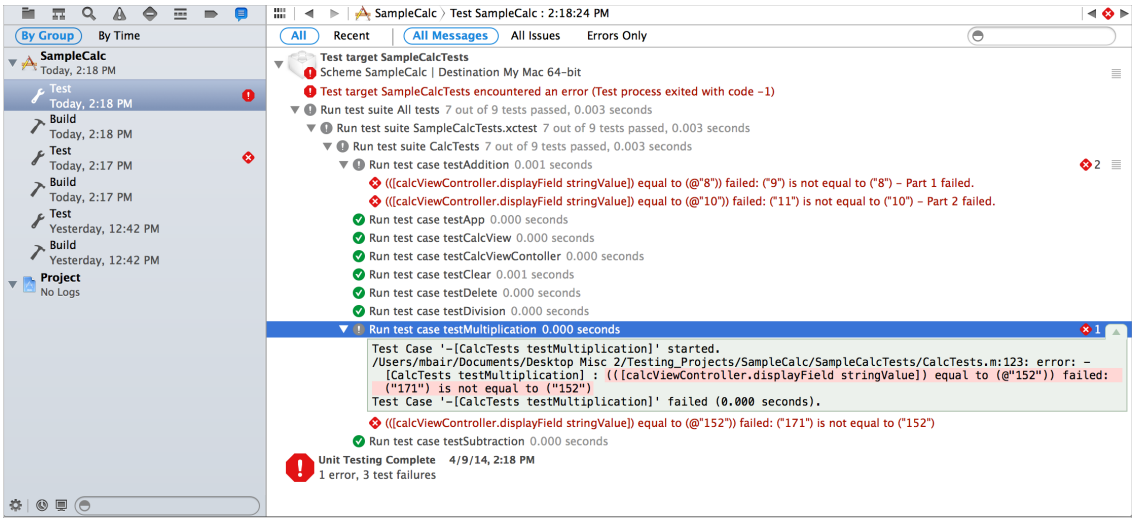
If test methods are collapsed into their respective class, or test classes into the test bundles, the indicator reflects the aggregate status of the enclosed tests. In this example, at least one of the tests in the `BasicFunctionsTests` class has signaled a failure.



- In the source editor, you can view pass/fail indicators and debugging information.

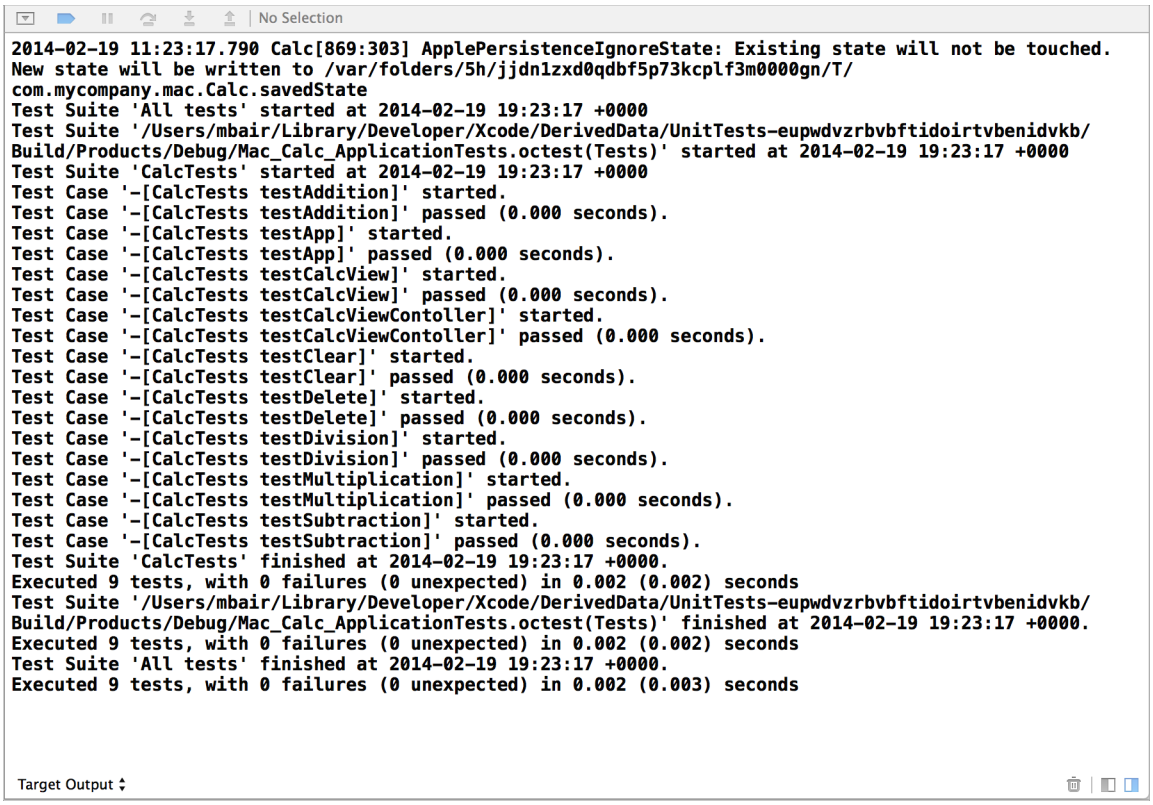


- In the log navigator, you can view the associated failure description string and other summary output. By opening the disclosure triangles, you can drill down to all the details of the test run.



Note: In addition to the disclosure triangles on the left of the item entries, the small icon to the right of a test failure item can be expanded to show more information, as you can see in the displayed `testMultiplication` failure above.

- The debug console displays comprehensive information about the test run in a textual format. It's the same information as shown by the log navigator, but if you have been actively engaged in debugging, any output from the debugging session also appears there.

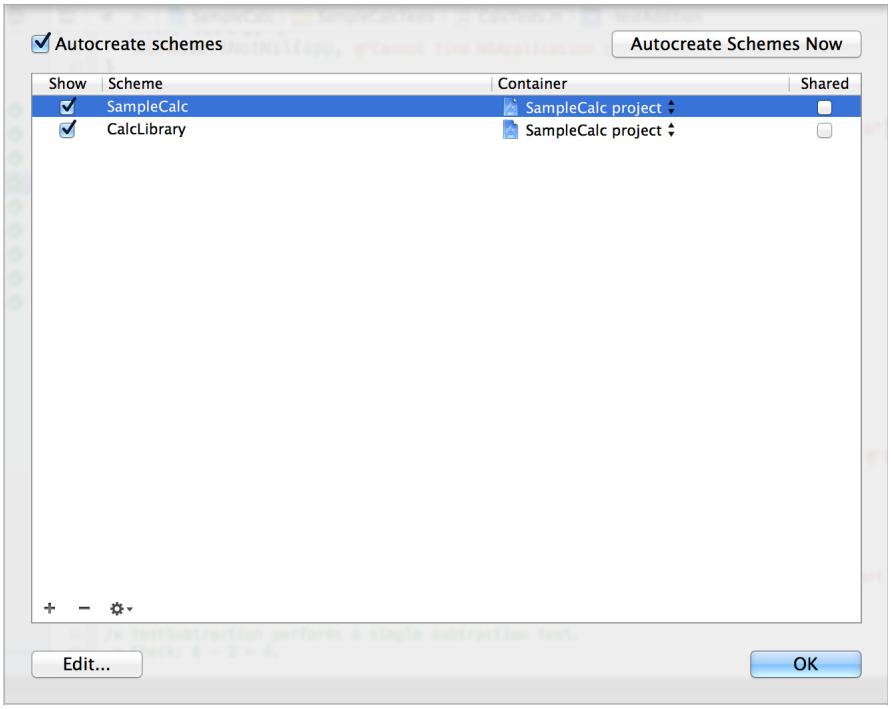


Working with Schemes and Test Targets

Xcode schemes control what the Build, Run, Test, and Debug menu commands do. Xcode manages the scheme configuration for you when you create test targets and perform other test system manipulations with the test navigator—for example, when you enable or disable a test method, test class, or test bundle. Using Xcode Server and continuous integration requires a scheme to be set to Shared using the checkbox in the Manage Schemes sheet, and checked into a source repository along with your project and source code.

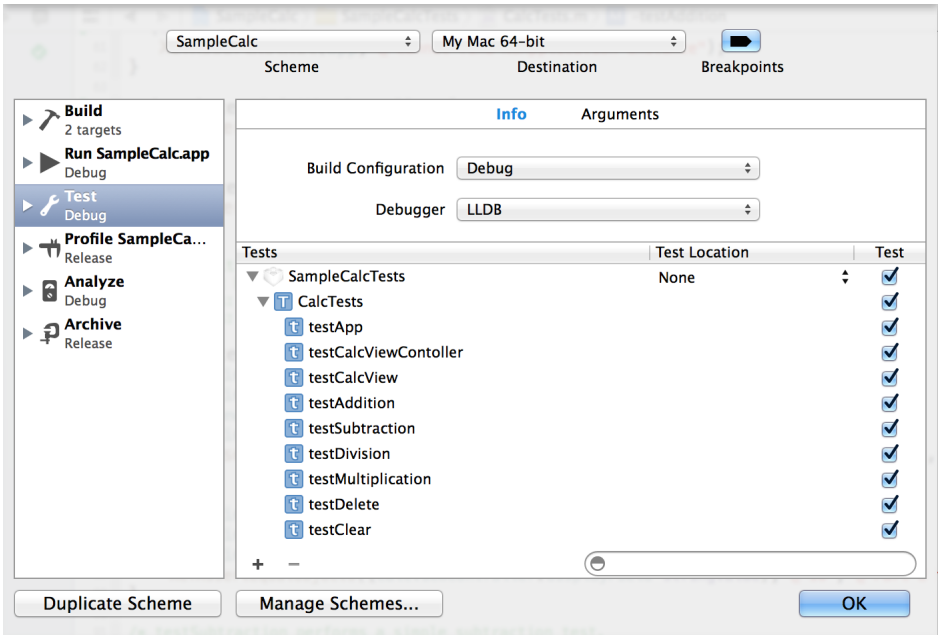
To see the scheme configuration for your tests:

1. Choose the Scheme menu > Manage Schemes in the toolbar to present the scheme management sheet.



In this project there are two schemes, one to build the app and the other to build the library/framework. The checkbox labeled Shared on the right configures a scheme as shared and available for use by bots with Xcode Server.

2. In the management sheet, double-click a scheme to display the scheme editor. A scheme's Test action identifies the tests Xcode performs when you execute the Test command.



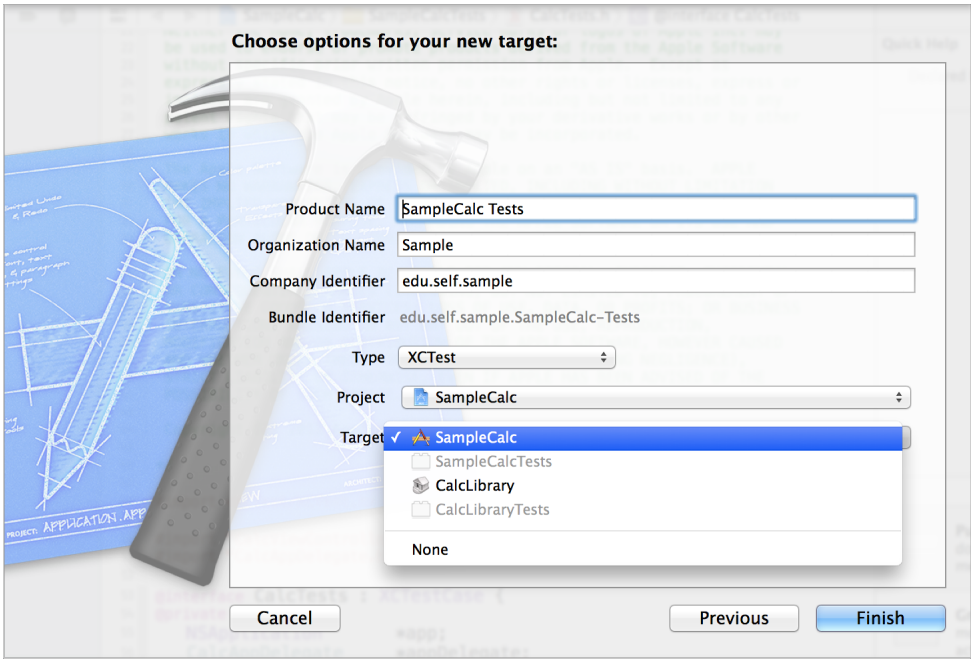
> **Note:** The test navigator and configuration/setup assistants associated with test targets, test classes, and test methods normally maintain all the scheme settings for you with respect to test operations.

Extensive information about using, configuring, and editing schemes is available in *Scheme Editor Help* and in the video presentation WWDC 2012: Working with Schemes and Projects in Xcode (408).

Build Settings—Testing Apps, Testing Libraries

App tests run in the context of your app, allowing you to create tests which combine behavior that comes from the different classes, libraries/frameworks, and functional aspects of your app. Library tests exercise the classes and methods within a library or framework, independent of your app, to validate that they behave as the library's specification requires.

Different build settings are required for these two types of test bundles. Configuring the build settings is performed for you when you create test targets by choosing the target parameter in the new target assistant. The target assistant is shown with the Target pop-up menu open. The app, `SampleCalc`, and the library/framework, `CalcLibrary`, are the available choices.



Choosing `SampleCalc` as the associated build product for this test target configures the build settings to be for an app test. The app process hosts the execution of your tests; tests are executed after receiving the `applicationDidFinishLaunching` notification. The default Product Name, "SampleCalc Tests," for the test target is derived from the `SampleCalc` target name in this case; you can change that to suit your preferences.

If you choose `CalcLibrary` as the associated build product instead, the target assistant configures the build settings for a library test. Xcode bootstraps the testing runtime context, the library or framework, and your test code is hosted by a process managed by Xcode. The default Product Name for this case is derived from the library target ("CalcLibrary Tests"). You can change it to suit your preferences just as with the app test case.

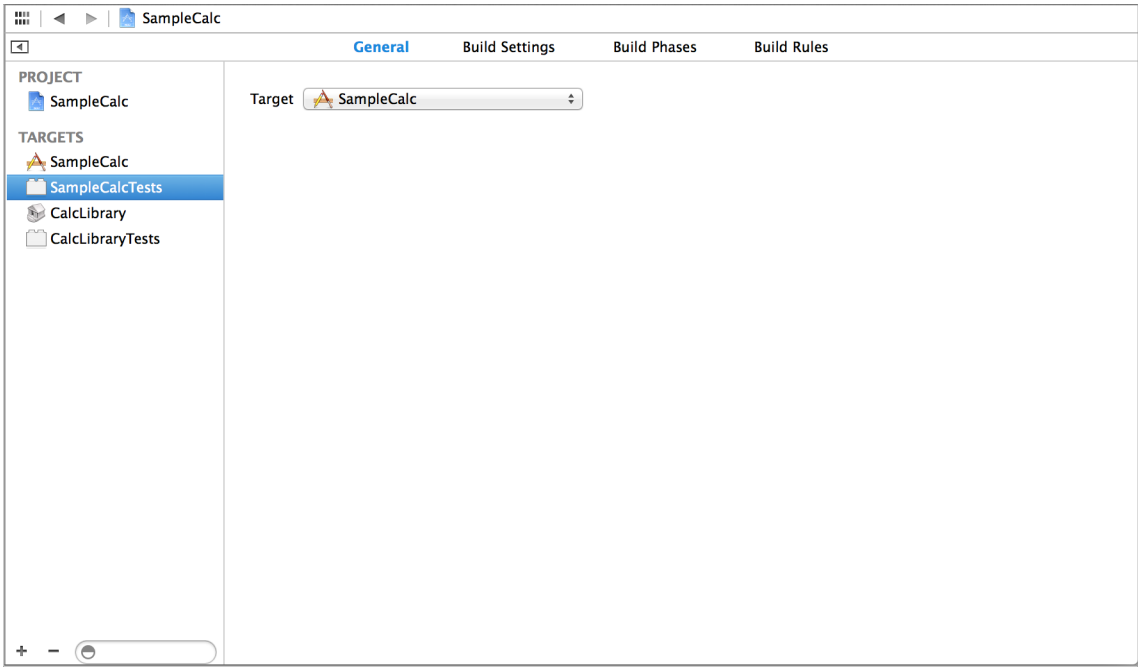
Build Setting Defaults

For most situations, choosing the correct test target to build product association is all you need to do to configure build settings for app and library tests. Xcode takes care of managing the build settings for you automatically. Because you might have a project that requires some complex build settings, it is useful to understand the standard build settings that Xcode sets up for app tests and library tests.

The `SampleCalc` project serves as an example to illustrate the correct default settings.

1. Enter the project editor by clicking the `SampleCalc` project in the project navigator, then select the `SampleCalcTests` app test target.

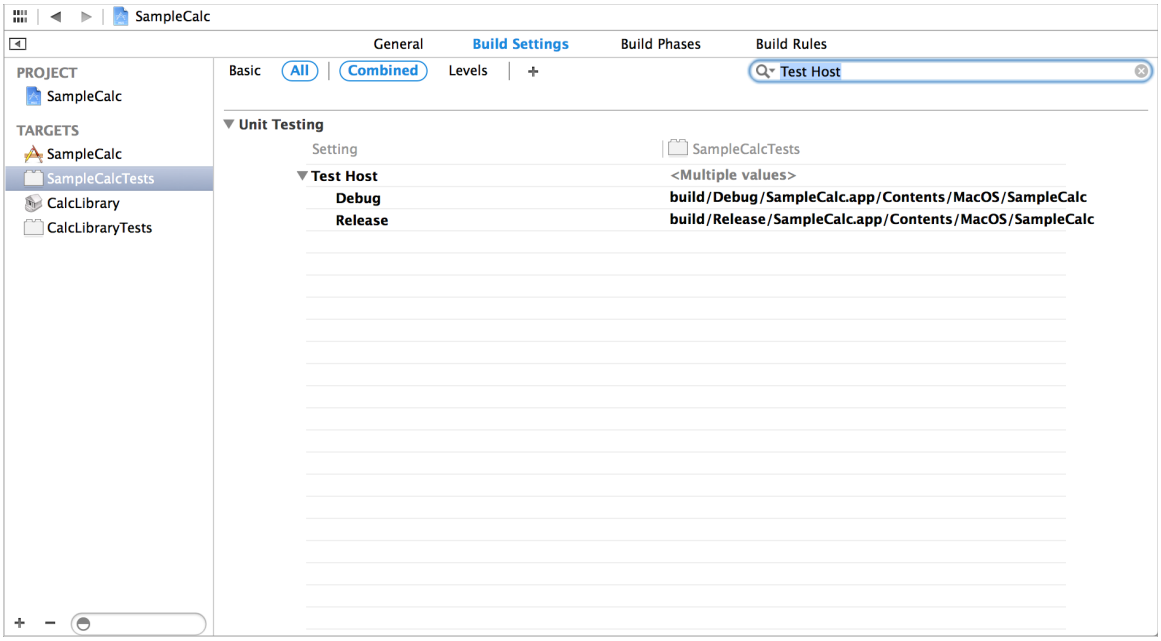
   In the General pane in the editor, a Target pop-up menu is displayed. The pop-up menu should show the `SampleCalc` app as target.



   You can check that the build settings are correct for the `SampleCalcTests` target.

2. Click Build Settings, then type `Bundle Loader` in the search field. The app tests for `SampleCalc` are loaded by the the `SampleCalc` app. You'll see the path to the executable as customized parameters for both Debug and Release builds.

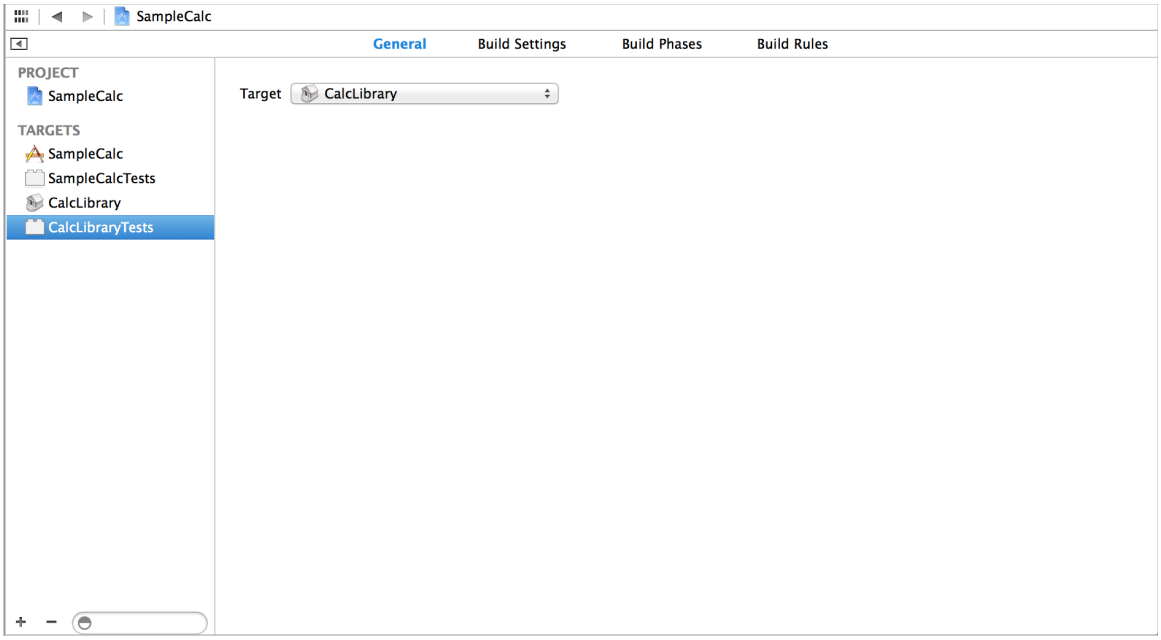
   The same paths appear if you search for `Test Host`, as shown.



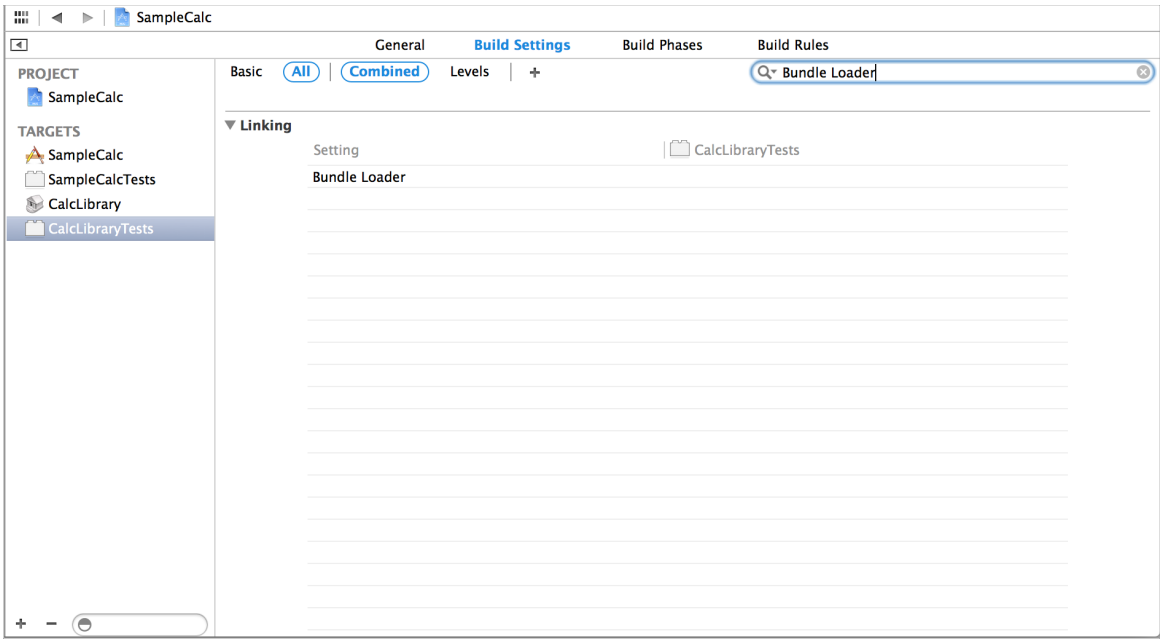
   The calculator library target of this project is named `CalcLibrary` and has an associated test target named `CalcLibraryTests`.

3. Select the `CalcLibraryTests` target and the General pane.

   The target is set to the `CalcLibrary` target.



4. Just as with the app test target, you can check for `Bundle Loader` and `Test Host` build settings using the Build Settings pane, shown below with the search set for `Bundle Loader`.



In the case of the library test target, the Bundle Loader and Test Host items have no associated parameters. This indicates that Xcode has configured them with its defaults, which is the correct configuration.

Testing Destinations

App tests run in OS X, iOS (on devices), and iOS Simulator. Library tests run on OS X and iOS Simulator. Xcode Server can run tests on a Mac with OS X Server by appropriately configuring the source repository, shared schemes, and bots. For details on setting up an Xcode Server run destination, see Automating the Test Process with Continuous Integration (page 49).

Debugging Tests

All of the standard Xcode debugging tools can be used when executing tests.

Test Debugging Workflow

The first thing to determine is whether the problem causing the failure is a bug in the code that you are testing or a bug in the test method that is executing. Test failures could point to several different kinds of issues—with either your assumptions, the requirements for the code that is being tested, or the test code itself—so debugging tests can span several different workflows. However, your test methods are typically relatively small and straightforward, so it's probably best to examine first what the test is intended to do and how it is implemented.

Here are some common issues to keep in mind:

- Is the logic of the test correct? Is the implementation correct?

  It's always a good idea to check for typos and incorrect literal values that you might be using as the reference standard that the test method is using as a basis of comparison.

- What are the assumptions?

  For example, you might be using the wrong data type in the test method, creating a range error for the code you're testing.

- Are you using the correct assertion to report the pass/fail status?

  For example, perhaps the condition of the test needs `XTCAssertTrue` rather than `XCTAssertFalse`. It's sometimes easy to make this error.

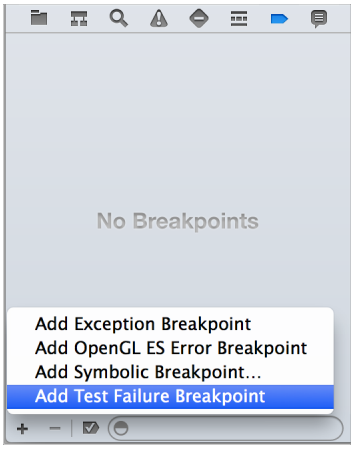
Presuming that your test assumptions are correct and the test method is correctly formed, the problem then has to be in the code being tested. It's time to locate and fix it.

Test Specific Debugging Tools

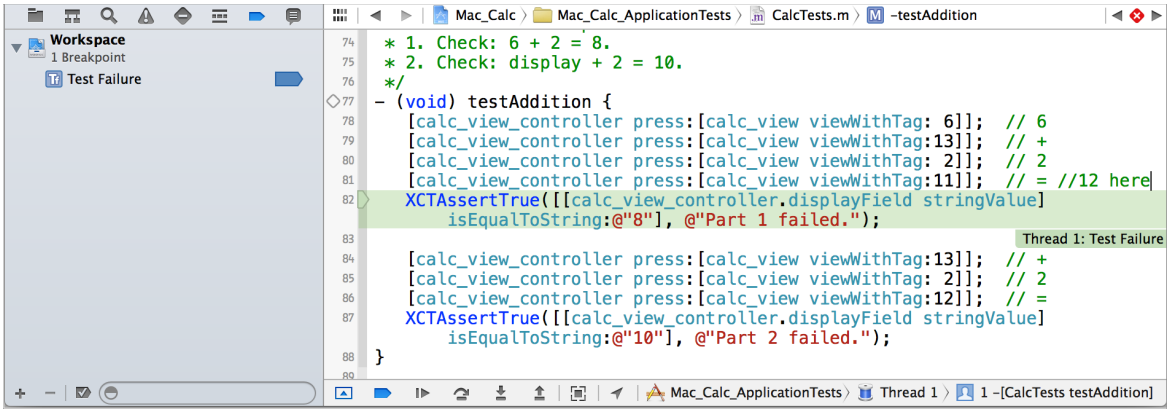
Xcode has a few special tools designed specifically to aid you in locating and debugging code when using tests.

Test Failure Breakpoint

In the breakpoint navigator, click the Add button (+) and choose Add Test Failure Breakpoint to set a special breakpoint before starting a test run.



This breakpoint stops the test run when a test method posts a failure assertion. This gives you the opportunity to find the location where a problem exists quickly by stopping execution right after the point of failure in the test code. You can see in this view of the `testAddition` test method that the comparison string has been forced to assert a failure by setting the reference standard for comparison to the wrong string. The test failure breakpoint detected the failure assertion and stopped test execution at this point.



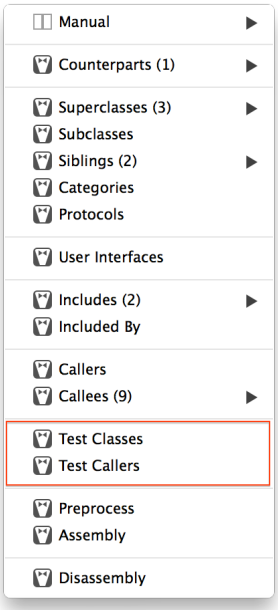
When a test run stops like this, you stop execution of the test. Then set a regular breakpoint before the assertion, run the test again (for convenience and time savings, you can use the Run button in the source editor gutter to run just this test), and carry on with debugging operations to fix the problem.

Using Project Menu Commands to Run Tests

Debugging test methods is a good time to remember the menu commands Project > Perform Action > Test Again and Project > Perform Action > Test. They provide a convenient way to rerun the last test method if you are editing code that is being fixed after a failure or to run the current test method you are working on. For details, see Using the Product menu (page 35). Of course, you can always run tests by using the Run button in the test navigator or in the source editor gutter, whichever you find more convenient.

Assistant Editor Categories

Two specialized categories have been added to the assistant editor categories to work specifically with tests.



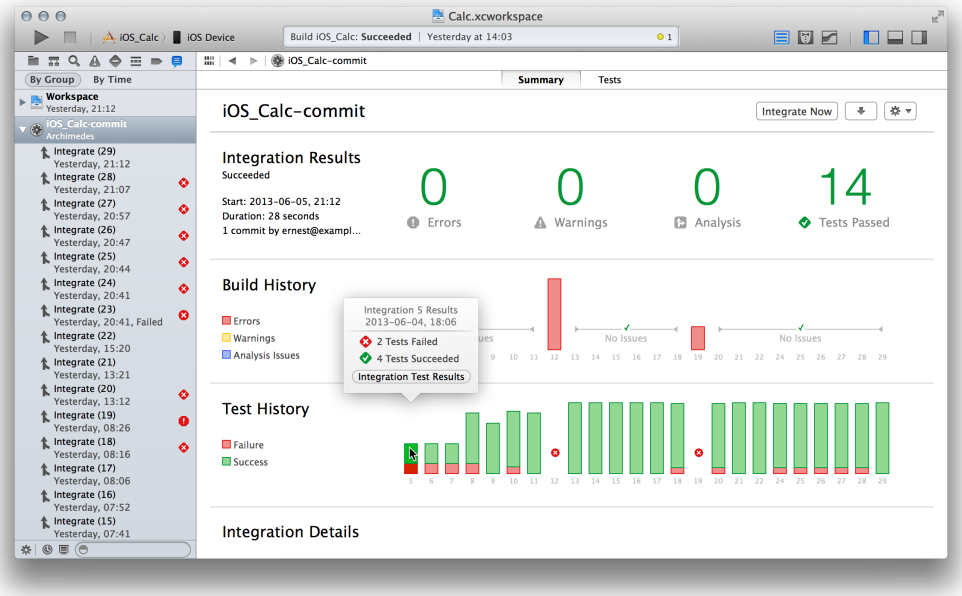
- **Test Callers category.** If you've just fixed a method in your app that caused a test failure, you might want to check to see whether the method is called in any other tests and whether they continue to run successfully. With the method in question in the source editor, open the assistant editor and choose the Test Classes category from the menu. A pop-up menu will allow you to navigate to any test methods that call it so you can run them and be sure that no regressions have been created by your fix.
- **Test Classes category.** This assistant editor category is similar to Test Callers but shows a list of classes which have test methods in them that refer to the class you are editing in the primary source editor. It's a good way to identify opportunities for adding tests, for example, to new methods you haven't yet incorporated into test methods that call them just yet.

Exception Breakpoints When Testing

Typically, an exception will stop test execution when caught by an exception breakpoint, so tests are normally run with exception breakpoints disabled in order that you locate inappropriately thrown breakpoints when they're triggered. You enable exception breakpoints when you're homing in a specific problem and want to stop tests to correct it.

Automating the Test Process with Continuous Integration

In addition to running tests interactively during development, take advantage of automating test runs using a server. This chapter describes using OS X Server and the continuous integration feature of Xcode Server to enhance and extend your testing.



Server-Based Testing with Continuous Integration

The Xcode testing capabilities, used interactively, ensure that your code stays on track with respect to its specified requirements and ensure that bugs which develop are easy to find and fix. A suite of fast-running functional tests proofs your work as you go and ensures solid app foundations that you can build upon efficiently and confidently.

That said, successful development projects tend to grow beyond the bounds of a single developer to implement and maintain. Like source management, automated testing on a server provides benefits by allowing your development effort to scale to the needs of a team smoothly and efficiently. Xcode supports a server-based continuous integration workflow through Xcode Server. Xcode Server, available in OS X Server, automates the integration process of building, analyzing, testing, and archiving your app.

Here are some benefits of using server-based testing:

- Using a server for offline build and test frees your development system to do implementation and debugging, particularly in the case when the full suite of tests takes a long time to run.
- All members of the development team run the same tests on the server by using the same scheme, thus enhancing test consistency. A server also makes build products available to the entire team, just as with build and test reports.
- You can adjust scheduling to both the project needs and your team needs. For instance, test runs can start when any member of the team commits new work to the source management system, or periodically, at set times. Test runs can also be started manually whenever required.
- The server runs the tests time after time, in exactly the same way. The reportage from the server benefits you and your team by giving you over time a picture of build issues, build warnings, and test resolutions.
- Your projects can be tested on many more destinations, automatically—and with more economy—than on a manually run testing system. For example, you can have an arbitrary number of iOS devices connected to the server, and with a single configuration, the system can build and test the libraries, apps, and tests on all of them, and in multiple versions of iOS Simulator as well.

Note: Two constraints apply when you're working on projects that span iOS and OS X:

- OS X projects are built and tested on the installation of OS X that the server is running.
- iOS and OS X projects require separate schemes and bots.

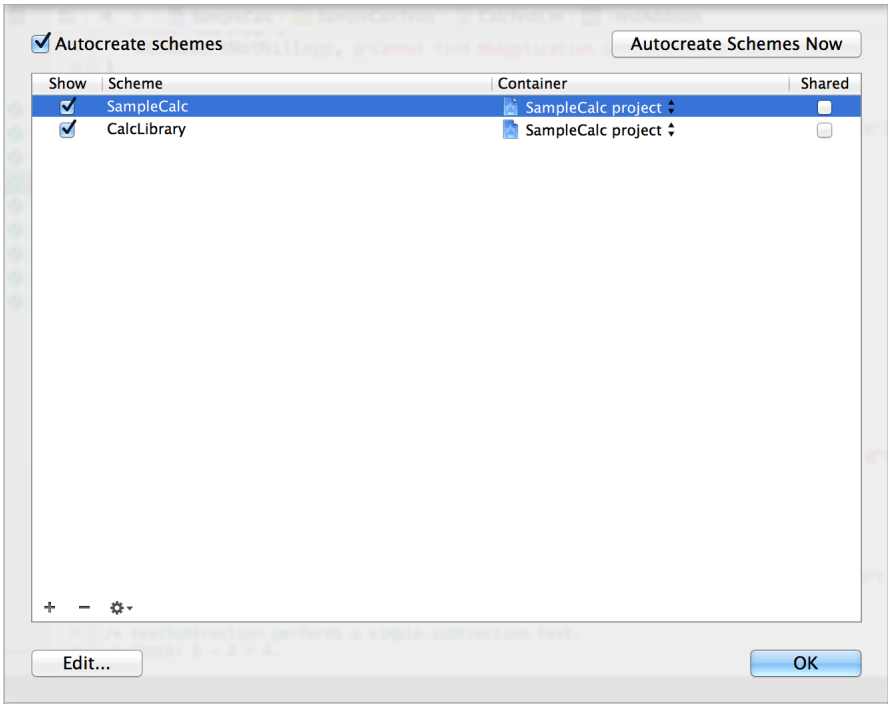
Overview—Using Xcode Server and Continuous Integration

Using Xcode Server and the continuous integration workflow is designed to be seamless and transparent to your interactive development work. Most of the effort involved in using server-based continuous integration occurs in the setup and configuration of the server and your project, briefly described here. There are only three high-level configuration tasks required to start using Xcode Server and the continuous integration system. After that the tasks become the ongoing monitoring of results and occasional maintenance. This overview provides a brief summary of these tasks with links to detailed information about them available in *Xcode Continuous Integration Guide*.

Shared Schemes, Bots, and Integrations

First, it's good to know a few key terms used when discussing Xcode Server.

Shared schemes. You should already be familiar with the concept of a scheme in Xcode; it is a recipe for building your project. It defines which targets are going to build, what dependent targets need to be built, and what build settings will be passed to the build process. Because a server is going to build your code, it needs access to this information. To do this, you configure a setting on your scheme to make it a *shared scheme* and check it into the source repository. You can do this manually on any scheme you've defined for your project using the Scheme Manager sheet by selecting the Shared checkbox to the right of the scheme in the list.



> **Note:** In most situations, Xcode automatically configures a shared scheme for you when you create a bot.

Bots. From your development Mac with Xcode, you create *bots* that run on a separate server, where they perform these integrations. Bots are a configuration for analyzing, building, testing, and archiving your project on a schedule. Bots help ensure that your product is always in a releasable state—and when there's a failure, the service notifies you or the person whose code change caused the failure.

Integrations. Bots perform *integrations*; an integration is a single run of a bot.

Configuration

There are only three high-level tasks required to start using Xcode Server and the continuous integration system.

Install and Set Up Xcode Server

Even if you've never set up a server before, you should find this straightforward and easily accomplished. You need to install OS X Server and configure Xcode Server on the intended machine. You'll also need to install Xcode on the server as well.

> **Note:** Be sure that the versions of OS X, OS X Server, and Xcode running on the server machine are all compatible.

Detailed instructions for installing OS X Server and configuring Xcode Server are described in Install OS X Server and Configure the Xcode Service in *Xcode Continuous Integration Guide*.

Connect Xcode Server to Source Code Repositories

Your bots must have access to the project's source code repositories. Xcode Server supports two popular source control systems: Git and Subversion. You can use Git and Subversion repositories hosted on remote servers, and you can host and use Git repositories on the server with OS X Server installed. The simplest case for setting up your remote repository is when you are creating a new project. In Xcode, you choose the option to create a git repository and choose the server from the "Create git repository on" pop-up menu in the new project setup workflow.

There are several other ways to set up your source repository, depending upon the project you are working with, its state with respect to source control, and so forth. Detailed instructions for setting up source repositories for all these situations are described in Enable Access to Your Source Code Repositories in *Xcode Continuous Integration Guide*.

Choose Product > Create Bot to Configure and Run Bots

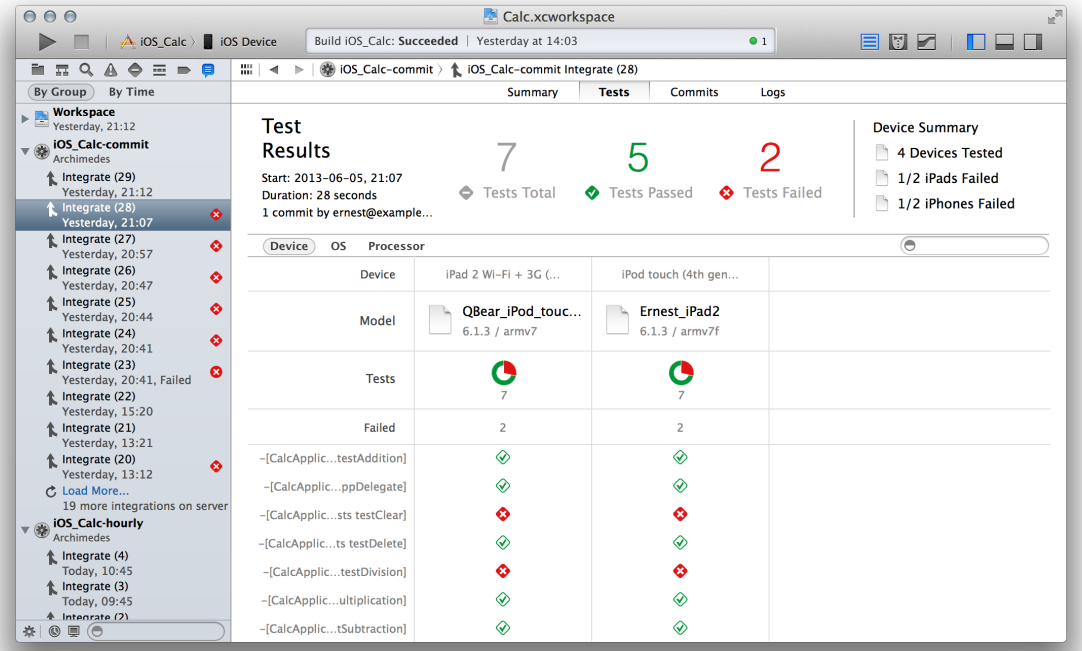
Bots are the heart of the automated workflow; they build and test products with your project's schemes. You create and schedule bots on your development system in Xcode to run either periodically or each time you commit source code. You can also configure bots to notify you about the success or failure of their integrations using emailed reports.

For details about creating and running bots, see Configure Bots to Perform Continuous Integrations in *Xcode Continuous Integration Guide*.

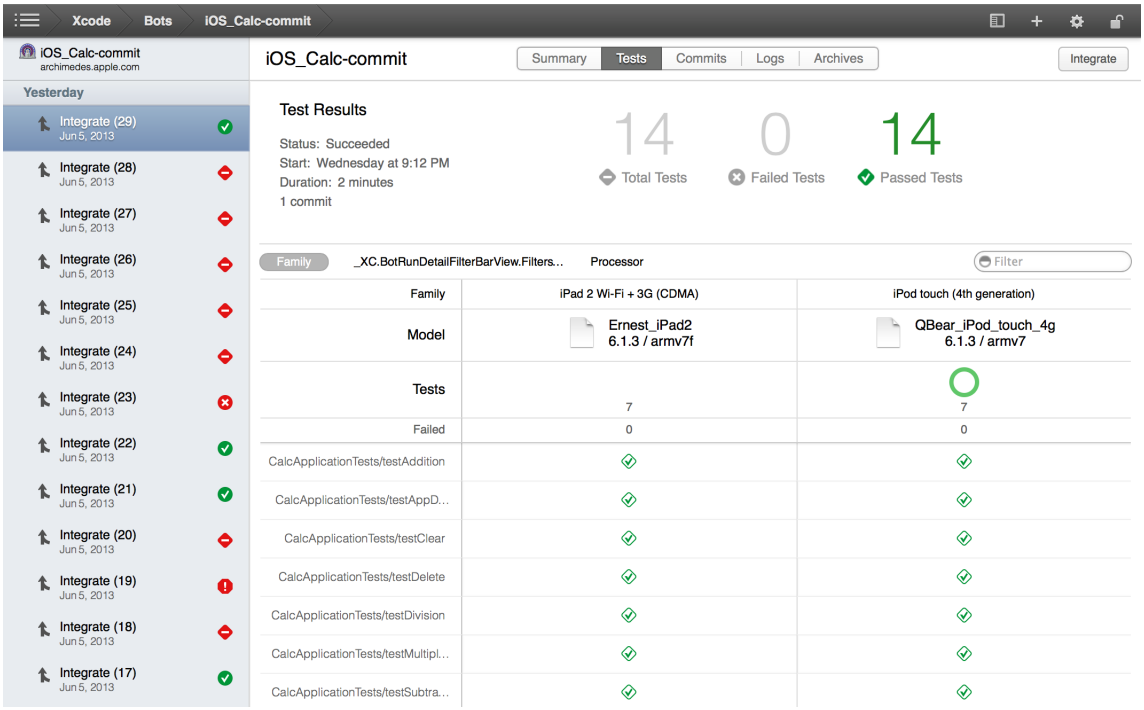
Continuous Integration Workflow

After you've completed the three configuration tasks, Xcode Server and the continuous integration system are up and running. These are designed to be a transparent addition to your interactive development workflow. Depending upon how you have configured bots to perform integrations, you perform your code development and local, interactive test tasks in the same way you have in the past. You should commit your work to the source repository as you normally would—when you reach stopping points, complete milestones, and so forth.

Your bots run on the server, performing the integrations as you've configured them, and return reports. In Xcode you use the log navigator to manage bots, view the test results, read the integration logs, initiate integrations, and download product builds and archives. For example, here is one of the available test result views using the log navigator:



You can also use a web browser to view the tests and reports.



To learn about all the different features for managing bots and seeing integration results, see Manage and Monitor Bots from the Log Navigator in *Xcode Continuous Integration Guide* and Manage and Monitor Bots from a Web Browser in *Xcode Continuous Integration Guide*.

Writing Testable Code

The Xcode integrated support for testing makes it possible for you to build test bundles to support your development efforts in a variety of ways. You can use tests to detect potential regressions in your code, to spot the expected successes and failures, and to validate the behavior of your app. Testing improves the stability of your code by ensuring that objects behave in the expected ways.

Of course, the level of stability you achieve through testing depends on the quality of the tests you write. Likewise, the ease of writing good tests depends on your approach to writing code. Writing code that is designed for testing helps ensure that you write good tests. Read the following guidelines to ensure that your code is testable and to ease the process of writing good tests.

Guidelines

- **Define API requirements.** It is important to define requirements and outcomes for each method or function that you add to your project. For requirements, include input and output ranges, exceptions thrown and the conditions under which they are raised, and the type of values returned (especially if the values are instances of classes). Specifying requirements and making sure that requirements are met in your code help you write robust, secure code.
- **Write test cases as you write code.** As you design and write each method or function, write one or more test cases to ensure that the API's requirements are met. Remember that it's harder to write tests for existing code than for code you are writing.
- **Check boundary conditions.** If a parameter for a method must have values in a specific range, your tests should pass values that include the lowest and highest values of the range. For example, if a procedure has an integer parameter that can have values between `0` and `100`, inclusive, the test code for that method should pass the values `0`, `50`, and `100` for the parameter.
- **Use negative tests.** Negative tests ensure that your code responds to error conditions appropriately. Verify that your code behaves correctly when it receives invalid or unexpected input values. Also verify that it returns error codes or raises exceptions when it should. For example, if an integer parameter must have values in the range `0` to `100`, inclusive, create test cases that pass the values `–1` and `101` to ensure that the procedure raises an exception or returns an error code.
- **Write comprehensive test cases.** Comprehensive tests combine different code modules to implement some of the more complex behavior of your API. Although simple, isolated tests provide value, stacked tests exercise complex behaviors and tend to catch many more problems. These kinds of tests mimic the

behavior of your code under more realistic conditions. For example, in addition to adding objects to an array, you could create the array, add several objects to it, remove a few of them using different methods, and then ensure that the set and number of remaining objects are correct.

- **Cover your bug fixes with test cases.** Whenever you fix a bug, write one or more tests cases that verify the fix.

Command-Line Testing

Using Xcode command-line tools, you can script and automate both the building and testing of your project. Use this capability to take advantage of existing build automation systems.

Using xcodebuild to Run Tests

The `xcodebuild` command-line tool drives tests just like the Xcode IDE and the Xcode service in OS X Server. Run `xcodebuild` with the buildaction `test`. Specify different destinations with the `-destination` argument. For example, to test MyApp on the local OS X "My Mac 64 Bit," specify that destination and architecture with this command:

```
> xcodebuild test -project MyAppProject.xcodeproj -scheme MyApp -destination
'platform=OS X,arch=x86_64'
```

If you have development-enabled devices plugged in, you can call them out by name or id. For example, if you have an iPod touch named "Development iPod touch" connected that you want to test your code on, you use the following command:

```
> xcodebuild test -project MyAppProject.xcodeproj -scheme MyApp -destination
'platform=iOS,name=Development iPod touch'
```

Tests can run in iOS Simulator, too. Use the simulator to target different form factors and OS versions easily. For example:

```
> xcodebuild test -project MyAppProject.xcodeproj -scheme MyApp -destination
'platform=iOS Simulator,name=iPhone,OS=7.0'
```

The `-destination` arguments can be chained together, letting you issue just one command to perform integrations across the destinations for the specified shared scheme. For example, the following command chains the previous three examples together into one command:

```
> xcodebuild test -project MyAppProject.xcodeproj -scheme MyApp
-destination 'platform=OS X,arch=x86_64'
```

```
-destination 'platform=iOS,name=Development iPod touch'
-destination 'platform=iOS Simulator,name=iPhone,OS=7.0'
```

If any tests fail, `xcodebuild` returns a nonzero exit code.

Those are the essentials of what you need to know about running tests from the command line. For more detailed information on `xcodebuild`, use `man` in a Terminal app window.

```
> man xcodebuild
```

Transitioning from OCUnit to XCTest

XCTest is the new testing framework introduced with Xcode 5. XCTest is a modernized reimplementation of OCUnit, the previous-generation testing framework. XCTest offers better integration with Xcode and lays the foundation for future improvements to Xcode testing capabilities. Many of the functions in XCTest are similar to the ones in OCUnit.

OCUnit and XCTest Compatibility

Because OCUnit has been the basis of testing in Xcode since Xcode 2.1, there are many projects containing existing OCUnit-based tests. They continue to work because Xcode was designed to offer compatibility between OCUnit- and XCTest-based projects side by side.

Existing projects with OCUnit-based tests brought into Xcode 5 from earlier revisions of Xcode are compatible with both current versions of iOS and OS X as well as versions of iOS earlier than iOS 7 and versions of OS X earlier than OS X v10.8.

OCUnit-based tests run on both iOS 6 and iOS 7, so OCUnit is still valid to use when projects targeting older versions of iOS and OS X with Xcode 5.x. It's also a valid choice when you need to be able to bring projects back from Xcode 5 to earlier versions of Xcode for legacy app maintenance.

However, as of Xcode 6, XCTest supports creating tests for iOS 6 and later. For most developers, converting your projects to use XCTest is the right choice.

> **Note:** OCUnit is marked as deprecated as of Xcode 5.1.

Converting from OCUnit to XCTest

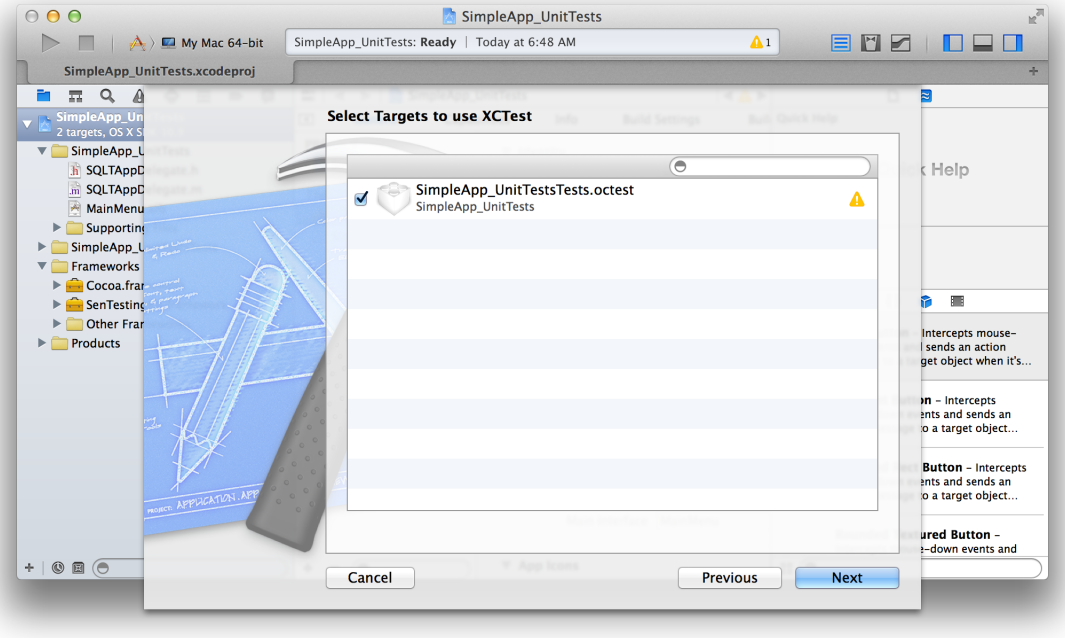
To convert from OCUnit to XCTest is a complex operation that includes updating source files that include test classes and revising project settings. Xcode includes a migrator to implement these changes for you, found by choosing Edit > Refactor > Convert OCUnit to XCTest. It is recommended that you use the migrator to update your project to use XCTest.

The Convert OCUnit to XCTest migrator works on a target-by-target basis. The migrator examines the targets you choose and lets you know whether they can be run with XCTest after the automatic conversion is performed.

> **Note:** It is recommended that you convert all targets in a project when doing the OCUnit to XCTest migration.

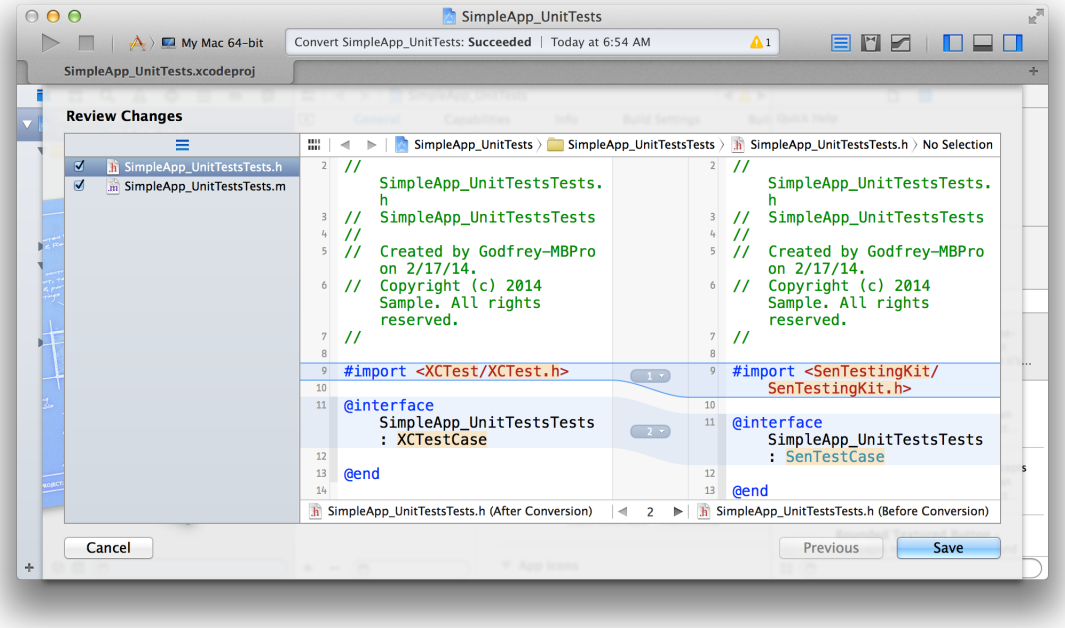
To update a project from OCUnit to XCTest:

1. Choose Edit > Refactor > Convert OCUnit to XCTest.
2. Click Next to move to the next sheet.
3. In the sheet that appears, select which test targets to convert.



4. To see whether or not a particular target will build with XCTest after conversion, click its name.
5. Click the Next button.

The assistant presents a FileMerge interface where you can evaluate the source changes on a file-by-file basis.



The updated source file is on the left, and the original is on the right. You can scan through all the potential changes and discard the ones you feel are questionable.

6. After you're satisfied that all the changes are valid, click the Save button.

   Xcode writes the changes to the files.

When the conversion has been completed, run the test target and check its outputs to be sure that the tests behave as expected. If any problems arise, see the Debugging Tests (page 45) chapter to determine what to do.

Converting from OCUnit to XCTest Manually

The Convert OCUnit to XCTest migrator may not be able to convert some projects successfully. In the event that it fails to convert your project (or if you wish to convert your tests to use XCTest manually), proceed as follows.

1. Add a new XCTest test target to the project.
2. Add the implementation files for your test classes and methods to the new test target.
3. Edit your test classes to `#import XCTest/XCTest.h` and use XCTest assertions in your test methods.

Using this methodology ensures that the project settings are properly configured for the XCTest test target. As when using the migrator, run the test target and check its outputs when you've completed the conversion to be sure that the tests behave as expected. If any problems arise, see the Debugging Tests (page 45) chapter to determine what to do.

Document Revision History

This table describes the changes to *Testing with Xcode*.

Date	Notes
2014-10-16	Updated for Xcode 6.1.
2014-06-13	New document that explains how to use the testing features in Xcode 5.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010