# Xcode Unit Testing Guide

**Tools & Languages: IDEs**

**2009-10-19**

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

# Contents

# Tables and Listings

## Adding Unit Tests to Your Projects 9

# Introduction

It is a familiar scenario to many developers. Major development is complete. The application has been tested. The team is fairly confident about the stability of the code. Then someone introduces that one last feature that breaks several seemingly unrelated features. Somehow, the new code has changed a fundamental assumption about the behavior of a code module and that change has now broken other modules. How do you validate your code in a way that will prevent this scenario from happening again? One way is through unit testing.

By adding appropriate unit tests to your projects, each engineer can verify that newly introduced code does not break any existing behavior. A unit test is simply a piece of code that exercises some part of your application. The unit test provides a specific input and expects your code to return a specific output. If your code returns an unexpected value, the unit test reports the discrepancy.

Third-party unit testing modules have been available for Xcode for some time. In addition, Xcode 2.1 now integrates some modules directly into the project environment. These modules provide the basic testing harness needed to build automated and repeatable test suites. This document shows you how to incorporate unit tests into your Xcode projects and also offers tips and guidance on how to get the most out of your tests in Mac OS X.

## Organization of This Document

This document includes the following article:

- "Adding Unit Tests to Your Projects" (page 9) shows you how to add unit testing capabilities to your Xcode projects.
- "Unit Test Guidelines" (page 21) provides some guidance on how to write unit tests effectively.

# Adding Unit Tests to Your Projects

Beginning with version 2.1, Xcode includes support for several open-source unit test modules. These add-on modules are shipped with the Xcode Tools and are installed automatically with Xcode. You can use these modules to add automated unit testing into your build process.

**iPhone OS Unit Testing:** To learn about unit testing iPhone applications, see *iOS Development Guide*.

The following sections show you how to write unit tests and how to modify your projects to execute those tests.

## Configuring Your Xcode Project

The first step in adding unit tests to your project is to create a new target in your project for building and running those tests. In Xcode, you create a build target of type "Unit Test Bundle". When built, this target compiles your test code and executes a shell script to run your tests. Upon completion of your tests, the Xcode console reports all failed tests as errors.

When you configure your test target, you need to decide how you want that target to behave. The unit test bundles that come with Xcode can be integrated with your executable in one of two ways. One technique is to configure your test target as a separate entity that you build and run independent of your main executable. The other is to add dependencies to your target that automatically build your executable and run the tests each time you build.

### Dependent Versus Independent Targets

When setting up your test code, you need to decide whether you want your test targets to be dependent or independent of your main executable. There are advantages and disadvantages to both techniques, which are listed in Table 1.

**Table 1** Pros and cons of target configurations

| Target type | Pros | Cons |
| --- | --- | --- |
| Independent | Easier to set up. Run only when wanted. | Must be run manually, which can lead to infrequent use. |
| Dependent | Easier to run tests automatically. Still allows option to build main executable separately. | Require some additional steps to set up. |

If you are writing an Objective-C program, the extra steps required to configure a dependent target are minimal. If you are writing a C or C++ program though, configuring a dependent target is a little more involved but still relatively straightforward.

The important decision to make is how often you want to run your test suite. If engineers are running the tests regularly, they are going to get a report of any errors with each build. If they build regularly, it can be easier to track down the changes that caused the problem.

Another decision to make is which tests you want to run during each build cycle. If you have several thousand tests, you might want to run them only at specific checkpoints, such as immediately before committing the changes to a repository, and not for every build. To specify which tests you want to run, you specify the name of the test suite in a command-line argument to the custom executable. For more information, see "Running Your Tests" (page 14).

## Creating Your Test Target

Regardless of whether you are configuring for dependent or independent targets, the initial set up process is identical:

1. Open your Xcode project.
2. Select Project > New Target to display the New Target assistant.
3. Select the desired target type:
   - If you are writing an Objective-C program, select the Cocoa > Unit Test Bundle target.
   - If you are writing a C/C++ program, select the Carbon > Unit Test Bundle target. You do not have to use the Carbon libraries to use this target.
4. Specify a name for your target and click Finish.

When you create the target, it is initially configured to be an independent target. If that is what you wanted, you can add your source code and test cases to the target and begin building. If you want to make it a dependent target, you must instead do the following:

- You must make your target dependent on the target for your main executable. (See "Making Your Target Dependent on the Main Executable" (page 10).)
- If your program is written in C or C++, you must add code to initialize your tests from within your application. (See "Executing Dependent Tests in C and C++" (page 11).) You do not need to do this for Objective-C based tests.

### Making Your Target Dependent on the Main Executable

To make your target dependent on your main executable, you must establish the dependency in your Xcode project:

1. Select your Unit Test Bundle target.
2. Open an inspector for the target.

3. Select the General tab.
4. Click the + button at the bottom of the window.
5. From the dialog that appears, select the target representing your main executable and click Add Target.

If you are building an application, you must also specify the path to your application's executable file in both the Bundle Loader and Test Host build settings. For example, to set the path to the executable of `MyApp.app`, you would do the following:

1. Select your Unit Test Bundle target.
2. Open an inspector for the target.
3. Select the Build tab.
4. Select the Linking collection.
5. Assign the value `$(CONFIGURATION_BUILD_DIR)/MyApp.app/Contents/MacOS/MyApp` to the Bundle Loader setting.
6. Select the Unit Testing collection.
7. Assign the value `$(CONFIGURATION_BUILD_DIR)/MyApp.app/Contents/MacOS/MyApp` to the Test Host setting. Because they use the same value, you can also set the value of this setting to `$(BUNDLE_LOADER)`.

**Note:** If you are testing a framework or shared library, you would not specify a value for the Test Host setting.

By default, the Bundle Loader and Test Host settings have no values assigned to them. Setting the value for Bundle Loader tells the linker to treat the specified executable as an additional framework at link time. (Treating it this way helps prevent unresolved references to application classes when you build your test target.) Setting the value for Test Host tells the `RunUnitTests` script (executed during the final build phase) to launch the specified application and inject your test bundle into it.

After configuring your test target, you should make it the active target and add your test case source files to it. The next time you build, Xcode will examine the dependencies of your test target and build your main executable first, followed by your test target, and then run your tests. This process ensures that your main code is built and the corresponding tests are run.

**Note:** For dependent test targets, add only your test case source files to the target. Do not add any source files from your main executable to the test target. When the tests are run, the test target bundle is injected into the main executable, where it accesses any source code required for the tests.

## Executing Dependent Tests in C and C++

If you are using a dependent test target for C++ projects, there is some extra code you must add to your test project to run your tests. Because of the dynamic nature of Objective-C, the bundle for a dependent test target can be injected into an executable automatically. In C++, however, this is not possible. Instead, you must explicitly tell the CPlusTest framework when it is safe to run your test code. One way to do this in Carbon applications is to set up a timer whose handler initiates your test suite as soon as the application finishes

launching and is capable of handling events. When the timer fires, the corresponding handler can then run the suite of tests that have already been registered, logging any output from the tests. (For information on how to register test cases, see "Creating Test Cases for C++" (page 17).)

Listing 1 shows the definition of a class you can use to initiate a timer in a Carbon application. An instance of this class is constructed at initialization time (see Listing 2 (page 12)) with the express purpose of installing a timer. When the timer eventually fires, the `firedTimerBridge` and `firedTimer` methods are used to run the actual tests.

**Listing 1** Interface for RunTestsInTimer.h

```
#include <Carbon/Carbon.h>

class RunTestsInTimer {
protected:
    EventLoopTimerUPP timerUPP;
    EventLoopTimerRef timerRef;

    static void firedTimerBridge(EventLoopTimerRef inTimer, void *inUserData);
    void firedTimer(EventLoopTimerRef inTimer);

public:
    RunTestsInTimer();
    virtual ~RunTestsInTimer();
};
```

Listing 2 shows part of the implementation file for this class. At the top of the implementation file is code for creating a global instance of the class. Because it is not a pointer, the actual class instance is constructed at initialization time. During that time, the constructor installs a timer on the application's main event loop. The fire delay and interval of the timer are set to 0 so that the timer is fired as soon as possible after the event loop is running and ready to process events.

**Listing 2** Installing the timer

```
#include <CPlusTest/CPlusTest.h>
#include "RunTestsInTimer.h"

// Create a local instance of the class at init time.
RunTestsInTimer installTimer;

RunTestsInTimer::RunTestsInTimer() : timerUPP(NULL), timerRef(NULL)
{
    // Get the UPP for the static bridge method.
    timerUPP = NewEventLoopTimerUPP(RunTestsInTimer::firedTimerBridge);

    (void) InstallEventLoopTimer(GetMainEventLoop(), 0, 0, timerUPP, this,
&timerRef);
}

// Clean up the timer structures.
RunTestsInTimer::~RunTestsInTimer()
{
    if (timerRef != NULL)
    {
        RemoveEventLoopTimer(timerRef);
        timerRef = NULL;
    }
```

```
    if (timerUPP != NULL)
    {
        DisposeEventLoopTimerUPP(timerUPP);
        timerUPP = NULL;
    }
}
```

Listing 3 shows the code that is executed when the timer eventually fires. This is the code that runs the actual test suites. Test cases are usually registered automatically when they are created, so you do not have to register them with the `TestSuite` object directly. As a result, the following code simply retrieves the tests that have been registered and runs them.

**Listing 3** Running the tests

```
// Static method to bridge the call to the local instance.
void RunTestsInTimer::firedTimerBridge(EventLoopTimerRef inTimer, void
*inUserData)
{
    RunTestsInTimer *self = (RunTestsInTimer *)inUserData;

    self->firedTimer(inTimer);
}

void RunTestsInTimer::firedTimer(EventLoopTimerRef inTimer)
{
    if (inTimer == timerRef)
    {
        TestRun run;

        // Create a log for writing out test results
        TestLog log(std::cerr);
        run.addObserver(&log);

        // Get all registered tests and run them.
        TestSuite& allTests = TestSuite::allTests();
        allTests.run(run);

        // Log a final message.
       std::cerr << "Ran " << run.runCount() << " tests, " << run.failureCount()
 << " failed." << std::endl;

        // Clean up.
        RemoveEventLoopTimer(timerRef);
        timerRef = NULL;

        QuitApplicationEventLoop();
    }
}
```

## Adding Test Cases to Your Target

Once you have configured your new test target, you can begin adding test cases to that target. Test cases are small pieces of code that exercise pieces of code in your main executable and report the results. After executing the code, the test case analyzes the results returned by that code. If the results are what was expected, the test case succeeds; otherwise, it fails.

**Note:** If your test target is independent of your main executable, you must also add the source code you want to test to your test target. To do this, drag the source files being tested to the compile sources build phase of your test target. You do not need to do this for dependent test targets.

For information on creating test files for Objective-C code, see "Creating Test Cases for Objective-C" (page 14). For information on creating test files for C or C++ code, see "Creating Test Cases for C++" (page 17).

## Running Your Tests

As long as your target is configured appropriately, all you need to do to run your tests is build your test target. Regardless of whether your target is dependent or independent of your main executable, the final build phase of the test target runs a shell script to execute your tests. The shell script performs the required set up, runs the tests, and reports any errors back to Xcode.

If you want to run only a subset of your tests when you build your target, you must rewrite the build script that is executed by your test target. Test targets execute the `RunUnitTests` script by default. This script handles the execution of targets built against either the SenTestingKit or CPlusTest frameworks. To run a subset of tests, you must create your own custom script (based on the information in `RunUnitTests`) that calls the appropriate testing rig with the options you want.

For C++ test bundles, custom scripts must call the `CPlusTestRig` tool, located in the `/Developer/Tools` directory. The `-test` option for this tool lets you specify a specific test case or test suite. For more information on using this tool, see the man page for `CPlusTestRig`.

For Objective-C test bundles, your custom script must call the `otest` tool, located in the `/Developer/Tools` directory. The `-SenTest` option for this tool lets you specify a test suite (class), a specific test method, or all tests. For more information on using this tool, see the man page for `otest`.

**Note:** Creating custom execution scripts is beyond the scope of this document. See the `RunUnitTests` script and the man pages for the test rig you are using for information on their use.

# Creating Test Cases for Objective-C

Support for Objective-C test cases is provided by the SenTestingKit framework. To create Objective-C tests, you create one or more test case classes and fill them with individual test methods. Each test method implements a simple test case you want to execute. Test methods can also implement more complex test cases or call other test methods to execute suites of tests. The following sections show you how to create test cases for Objective-C.

## Creating Your Test Case Class

For Objective-C, your test case class is a subclass of SenTestCase. To create a new instance of this class in your Xcode project, do the following:

1. Select File > New File.
2. In the file type dialog, select Cocoa > Objective-C test case class and click Next.
3. Give your class a name and be sure to create the matching header file.
4. Add the class to your test target. Do not add it to the target for your main executable.
5. Click Finish.

When you're done, the header file for your new class should look similar to the following:

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

@interface MyTests : SenTestCase
{
}
@end
```

Your source file should simply contain the currently empty implementation for your class. You can now begin to add test cases to the implementation file.

## Writing Your Test Cases

The SenTestCase class provides a harness for calling the methods of your subclass automatically. As long as you follow some standard naming conventions for your test case methods, the only thing you have to do is write your tests. Your test case methods must follow these conventions:

- The name of the method must begin with the word `test`. For example, you could have methods called `testCase1`, `testMyBoundsChecking`, `testMyAlgorithm`.
- The method must take no parameters.
- The method must have a return type of `void`.

Thus, using the preceding criteria, the method implementation for the `testCase1` method would look like the following:

```
- (void) testCase1
{
    ...
}
```

Methods defined in this way are discovered dynamically and called automatically by the supporting test harness. You do not have to do anything to register your tests or call them explicitly.

In the body of your test case methods, you must construct any data structures you need to execute the test, run the test, and then release the data structures you created. If a test succeeds, the corresponding test case method should exit normally. If a test fails, you should report the failure using one of the macros defined by the SenTestingKit framework. For a list of basic macros, see "Commonly Used Macros" (page 16).

## Managing Common Objective-C Structures

If you have multiple test case methods that operate on the same basic data structures, there is no need to repeat the code for creating and destroying those data structures in each test case method. The SenTestCase class defines the `setUp` and `tearDown` methods for creating and destroying common data structures before and after each test case method is run. You can override these methods in your subclass and use them to create whatever data structures are used by your test methods. The basic implementation of these methods is as follows:

```
- (void) setUp
{
    // Create data structures here.
}

- (void) tearDown
{
    // Release data structures here.
}
```

At runtime, the `setUp` method is called immediately before each test case method. Similarly, the `tearDown` method is called immediately after to clean up the data structures. In your `tearDown` method, it is a good idea to reset the values of your data structures to a known initial state. For example, you might want to reset any object pointers to `nil` for safety.

## Commonly Used Macros

The SenTestingKit framework defines a number of assertion macros that you can use to determine the success or failure of a given test. These macros are defined in the `SenTestCase.h` header file of the framework. Some of the more commonly used macros are listed below:

```
STAssertNotNil(a1, description, ...)
STAssertTrue(expression, description, ...)
STAssertFalse(expression, description, ...)
STAssertEqualObjects(a1, a2, description, ...)
STAssertEquals(a1, a2, description, ...)
STAssertThrows(expression, description, ...)
STAssertNoThrow(expression, description, ...)
STFail(description, ...)
```

The *`description`* parameter of each macro lets you specify a human-readable string to be printed when the assertion fails. The string supports the same printf-style value substitution used by NSString, which you might use to print out the actual and expected values for the test. Listing 4 shows some examples of how to use some of these macros in your code.

**Listing 4** Macro examples in Objective-C

```
- (void) testObjectCreation
```

```
{
    MyObject* theObj = [[MyObject alloc] init];

    STAssertNotNil(theObj, @"Could not create instance of MyObject.");
}

- (void) testObjectsAreEqual
{
    NSNumber* theObj = [NSNumber numberWithInt:1];
    NSNumber* theObj2 = [NSNumber numberWithInt:1];

    STAssertEqualObjects(theObj, theObj2, @"Objects were not equal. Value 1: %d
 Value 2: %d", [theObj intValue], [theObj2 intValue]);
}
```

For more information about these macros, see the documentation that comes with the SenTestingKit framework, which is located in the framework's `Resources` directory.

# Creating Test Cases for C++

Support for C and C++ test cases is provided by the CPlusTest framework. You can use this framework to test C or C++ code for everything from BSD to Carbon applications. To create a test case, you have to do the following:

1. Create a new subclass of TestCase.
2. Define the methods that comprise your tests.
3. In your implementation file, register your tests by creating a local instance of your class for each test. See "Registering Your Tests" (page 19).

The following sections show you how to create test cases for your C and C++ code.

**Note:** Although you can test C code, the tests themselves must still be written in C++ and use the CPlusTest framework infrastructure.

## Creating Your Test Case Class

For C/C++, your test case class is a subclass of TestCase. To create a new instance of this class in your Xcode project, do the following:

1. Select File > New File.
2. In the file type dialog, select Carbon > C++ Test Case and click Next.
3. Give your class a name and be sure to create the matching header file.
4. Add the class to your test target. Do not add it to the target for your main executable.
5. Click Finish.

When you're done, the header file for your new class should look similar to the following:

```
#include <CPlusTest/CPlusTest.h>

class MyTests : public TestCase
{
public:
    MyTests(TestInvocation* invocation);
    virtual ~MyTests();
};
```

Your source file should contain default implementations for your constructor and destructor. You should not need to modify these implementations and should definitely not use them to create and destroy common data structures. Common data structures must be created using the `setUp` and `tearDown` methods; see "Managing Common C++ Structures" (page 18).

## Writing Your Test Cases

The TestCase class does not impose any naming conventions on your test case methods, although they must follow some calling conventions:

- The method must take no parameters.
- The method must have a return type of `void`.

Thus, using the preceding criteria, the method implementation for the `testCase1` method would look like the following:

```
void MyTests::testCase1()
{
    ...
}
```

In the body of your test case methods, you can construct any data structures you need to execute the test, run the test, and then release the data structures you created. If a test succeeds, the corresponding test case method should exit normally. If a test fails, you should report the failure using the `CPTAssert` macro.

## Managing Common C++ Structures

If you have multiple test case methods that operate on the same basic data structures, there is no need to repeat the code for creating and destroying those data structures in each test case method. The TestCase class defines the `setUp` and `tearDown` methods for creating and destroying common data structures before and after each test case method is run. You can override these methods in your subclass and use them to create whatever data structures are used by your test methods. The basic implementation of these methods is as follows:

```
void MyTests::setUp()
{
    // Create data structures here.
}

void MyTests::tearDown()
{
```

```
    // Release data structures here.
}
```

At runtime, the `setUp` method is called immediately before each test case method. Similarly, the `tearDown` method is called immediately after to clean up the data structures. In your `tearDown` method, it is a good idea to reset the values of your data structures to a known initial state. For example, you might want to reset any object pointers to `NULL` for safety.

## Registering Your Tests

Unlike Objective-C tests, which are discovered dynamically, C++ tests must be registered explicitly if you want them to run. To register your tests with the CPlusTest framework, you must create a new instance of your test class for every test case you want to run. You create these instances in the global scope, that is, defined outside of the scope of any methods.

When you create a new instance of you test case class, you must pass an object parameter of type `TestInvocation`. This object contains the details of the test you want to run, including the class name and method. To create this object, you can use the `TEST_INVOCATION` macro, which takes the target class and method names as parameters. The following example creates two tests for the same test case class. The first test case executes the `MyFirstTest` method of the MyTests class while the second executes the `MySecondTest` method.

```
MyTests test1(TEST_INVOCATION(MyTests, MyFirstTest));
MyTests test2(TEST_INVOCATION(MyTests, MySecondTest));
```

Creating the objects actually executes code to register the specified invocation information with the CPlusTest framework. Because you create these objects in the global scope, they are created during the initialization phase of the test executable and are thus available later when the framework wants to run your tests.

# Unit Test Guidelines

Xcode's integrated support for unit testing makes it possible for you to build test suites to support your development efforts in any way you want. You can use it to detect potential regressions in your code or validate the behavior of your application. These capabilities can add tremendous value to your projects. In particular, they can improve the stability of your code greatly by ensuring individual pieces of code behave in expected ways.

Of course, the amount of stability you receive from unit testing is highly dependent on the quality of the test cases you write. The following are some guidelines to think about when writing test cases:

- One test is infinitely better than no tests at all. One test ensures that your code compiles, links and can run.
- Whenever a bug is fixed, write one or more test cases to verify that the behavior remains fixed.
- Use dependent targets to make it easier to run tests regularly. Running tests regularly can help identify problems before they are committed to your source code repository.
- Check boundary conditions heavily. If the parameter of a method expects values in a specific range, your tests should pass in values that lie across that range. For example, if an integer parameter can have values between 0 and 100 inclusive, three variants of your test might pass in the values 0, 50, and 100 respectively.
- Use negative tests to be sure your code responds to error conditions appropriately. Verify that your code behaves appropriately when it receives invalid or unexpected input values. Verify that it returns errors or throws exceptions when it should. You might be surprised to find that a test you expected to fail actually succeeds. For example, if an integer parameter to a method can accept values in the range 0 to 100 inclusive, you might create tests that pass in the values -1 and 101.
- Write tests that combine different code modules to implement some of the more complex behaviors of your application. While simple, isolated tests do provide value, stacked tests that exercise complex behaviors tend to catch many more problems. These kinds of tests simulate the behavior of your code under more realistic conditions, which leads to the discovery of more realistic problems. For example, in addition to just adding objects to an array, you could create the array, add several objects to it, remove a few of those objects using several different methods, and then make sure the number of remaining objects is correct.
- Don't stress about unit tests. They are intended as a tool for ensuring good test coverage and memory management. Use them in that way to aid your development process.

# Document Revision History

This table describes the changes to *Xcode Unit Testing Guide*.

| Date | Notes |
| --- | --- |
| 2009-10-19 | Made technical corrections. |
| | Replaced BUILT_PRODUCTS_DIR with CONFIGURATION_BUILD_DIR in "Making Your Target Dependent on the Main Executable" (page 10). |
| 2008-05-02 | Updated for Xcode 3.0. |
| | Changed title from *Unit Testing Guide*. |
| 2005-06-06 | New document that explains how to incorporate unit tests into your development process using Xcode. |

# Index