UnitKit User Manual

Version 1.1• October, 2004 • James Duncan Davidson

This document doesn't asnwer the relevant question "What is unit testing and why should I care?" I'm leaving the answer to other documents that you can find on the web.

UnitKit is a unit test framework for Objective-C based projects.

Why on earth would the world need another unit testing framework for Objective-C, especially since there are already other respectable and useful frameworks for Objective-C, including OCUnit, ObjCUnit, and TestKit?

There are four parts to the answer:

- First, UnitKit strives to be less (or maybe more) than a clone of JUnit or SUnit. Instead of just copying the same basic thoughts to yet another language, it is an attempt to rethink how a basic testing framework can be put together that leverages what Objective-C is, and is not, as a language.

Xcode is the new Integrated Development Environment for Mac OS X introduced at WWDC2003. You can download it from the Apple Developer Connection website.

- Second, UnitKit is designed to work elegantly, from scratch, with the Xcode IDE. Xcode brings a lot of new ideas to the IDE table that are designed to keep the developer moving. Integrating unit testing into its way of working seems like a "Good Thing To Do."
- Third, UnitKit attempts to be as small and compact as possible while fulfilling the first two goals. Anything that isn't needed for those goals has (hopefully) been tossed out along the wayside.
- Fourth, UnitKit strives to have an excellent user experience. It installs easily and is easy to add to existing projects.

UnitKit will always strive to be as small and compact as possible given its mission. Features will be added, but only grudgingly. This isn't to say that minimalism should win at all costs—only that there shouldn't be anything built prematurely.

Installation

There are a variety of ways that UnitKit can be installed on your system. The easiest way is to use the installer package (`UnitKit.pkg`) which is distributed from the UnitKit website as part of a compressed disk image. The contents of this disk image are shown in Figure 1.

Figure 1

The UnitKit distribution disk image which contains the UnitKit installation package, source code, and a few other handy things.

When this package is installed, the following components are added to your system:

```
/usr/local/bin/ukrun
/Library/Frameworks/UnitKit.framework
```

As well, the various Xcode integration templates are added to:

```
/Library/Application Support/Apple/Developer Tools
```

The installer package was built so that it should respect symlinks. For example, if you have `/usr/local` symlinked to another disk, the installer should respect the link instead of clobbering it.

Building UnitKit

Use care when using the xcodebuild command in combination with setting the DSTROOT to the root directory (/). If you aren't careful, bad things can happen. See the xcodebuild man page for more information.

If you would like to build the various parts of UnitKit yourself, you can use the following command while in the UnitKit source directory:

```
$ sudo xcodebuild -target ukrun DSTROOT=/ install
```

This will build UnitKit and place everything into the correct location. However, you should be careful using this command as you will obliterate any current UnitKit installation.

Building UnitKit for GNUstep

To build UnitKit for GNUstep, use the following sequence of commands:

```
$ make
$ sudo make install
```

Uninstalling UnitKit

To uninstall UnitKit, you can remove all the various UnitKit files from their locations—or you can use the quickly script provided in the distribution:

```
$ ./uninstall
Removing /usr/local/bin/ukrun
Removing /Library/Frameworks/UnitKit.framework
Removing file templates from /Library/Application Support
...
Removing Package Receipt
```

This script will leave your system clean of UnitKit and will even clean up directories used by previous versions of UnitKit.

Using UnitKit, Step-by-Step

Now that UnitKit is installed and ready to use, it'd be good to know how to use it, huh?

One place to start is to look at a sample project. UnitKit ships with the Currency Converter project (from the Apple documentation as well as Learning Cocoa with Objective-C) in the Sample Projects folder of the distribution, shown in Figure 2. There are two versions of the project there—one in its original pre-test form and another in a form with unit tests.

To get a bit of practice adding tests to an existing project, open up the pre-test Currency Converter project. Once it's open, build and run the application to get an idea of what it does.

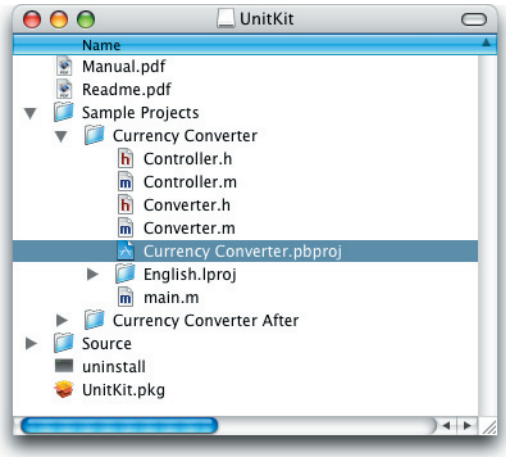
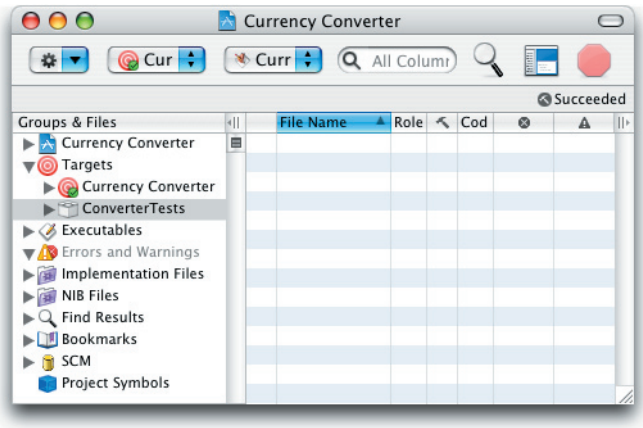


Figure 2

The location of the CurrencyConverter sample project within the distribution package.

Figure 3

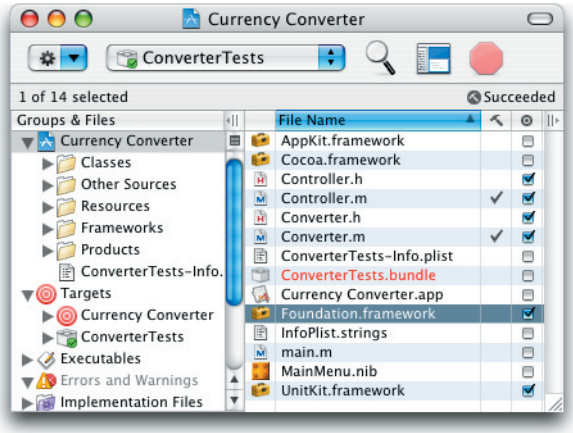
The CurrencyConverter project with the ConverterTests test bundle added to the Targets group



1. Add a new target to the project. Use the Project > New Target menu. In the New Target dialog box, select the Cocoa > UnitKit Bundle target type, and then click the Next button.
2. Give the bundle a descriptive name—something like `ConverterTests`. The target will be created and you'll see it appear in the Targets group, as shown in Figure 3.
3. Next, add the UnitKit framework to the project. Use the Project > Add Frameworks menu and select `/Library/Frameworks/UnitKit.framework`. When you reach the final dialog, be sure to add the framework on the ConverterTests target.
4. Now, add the classes from the Currency Converter application that will be tested. Make sure that the ConverterTests target is selected as the active target in the top left corner of the Xcode project window. Then select the check boxes of the Converter class files, `Converter.h` and `Converter.m`, as shown in Figure 4.

Figure 4

Selecting the files that are part of the ConverterTests target. Notice that you should select the project as a whole to see all of the files and make sure that you have the ConverterTests target selected.



It is odd that you'd have to select the Foundation.framework to be part of the ConverterTest target, but it's just a fact of life for now. Hopefully that will change in the future.

You'll probably see a warning about the UnitKit framework not being prebound. The warning is correct, but you don't have to worry about anything. In fact, you should really turn prebinding off in your application builds now that it is no longer needed (as of Mac OS X 10.3.4).

You'll also want to make sure that the `Foundation.framework` is selected as part of the ConverterTests target.

Build the target (⌘-B). At first, you won't notice much has happened except maybe that your hard drive made a bit of noise. To see what has happened behind the scenes, use the Build > Show Detailed Build Results Menu (⌘-Shift-B). Then, open up the split window to see the build transcript, as shown in Figure 5. Scroll down to the end of the transcript until you see the line:

```
Result: 0 classes, 0 methods, 0 tests, 0 failed
```

This line is output from `ukrun` and tells us that the test bundle was built and that as part of the build UnitKit attempted to run any tests in the bundle. There aren't any tests in the bundle yet. Let's fix that.

5. Use the File > New File menu. In the dialog box that appears, select the Objective-C UnitKit class entry. Name the class something reasonable like `ConverterTest`.

6. Now we're going to add a test method. Let's start out with a test failure. Modify `ConverterTest.m` to match the following:

```
#import "ConverterTest.h"

@implementation ConverterTest

- (void) testNothing
{
    UKFail();
}

@end
```

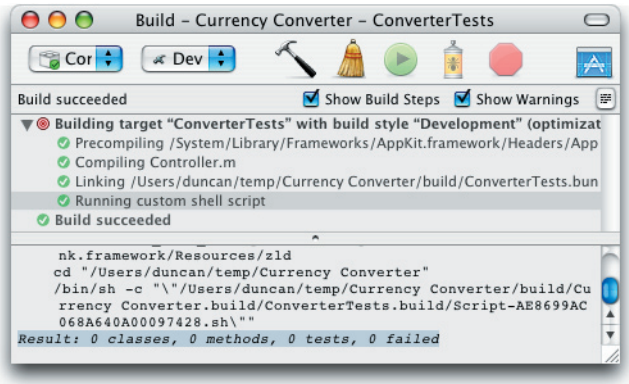
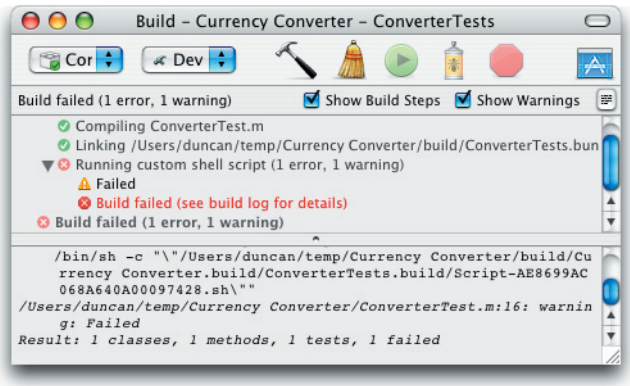


Figure 5

The detailed build window showing the various steps taken during the build of the target. The last shell script entry is ukrun's execution. Notice that no tests were executed as there weren't any found.

Figure 6

The detailed build window showing a test failure. Click on the yellow Failed entry and Xcode will take you to the failing assertion. Xcode 1.5 adds a double split window that you'll have to fidget with to get the build log to show on the bottom of the window.



The `UKFail()` declaration is an assertion that indicates that a test should be considered a failure. When this is called, UnitKit performs all the needed steps to report a test failure.

In Xcode versions 1.0 through 1.2, this is the extent of test failure reporting. However, in Xcode 1.5, this has changed and test failures will also appear in the "Errors & Warnings" smart group in the main project window.

Build the target again (⌘-B). This time you'll see something in the Build window, as shown in Figure 6. Click on the `UKFail` warning and Xcode will take you to the failing test in your code. Now you're starting to see the integration with Xcode in effect. Not so bad, eh?

Let's put a more realistic test in.

7. Replace the failing test we added in step 6 with the following (and don't forget the import statement):

```
#import "ConverterTest.h"
#import "Converter.h"

@implementation ConverterTest

- (void) testConverter
{
    float initialAmt = 27.30;
    float rate = 3.33;
    Converter *converter = [[Converter alloc] init];
    float convertedAmt = [converter
        convertAmount:initialAmt atRate:rate];
    UKFloatsEqual(90.90, convertedAmt, 0.01);
    [converter release];
}

@end
```

`UKFloatsEqual(a, b, delta)` is an assertion that passes if `a` and `b` are equal within the margin of error specified by `delta`. If the values are not equal, then a test failure is reported.

Now, build the target again (⌘-B). Depending on how you have Xcode setup, you might not see the Build window at all this time. What you will see is a build-succeeded icon in the top left of the project window.

This little step-by-step should have given you the basics of how to get started with UnitKit. Now, let's look at how UnitKit works behind the scenes.

How UnitKit Works

UnitKit follows many of the same conventions as other unit testing frameworks for both sanity purposes as well as familiarity. Here's the basic rules of the road as to how UnitKit works.

1. Any class in the test bundle that conforms to the `UKTest` protocol is automatically picked up for testing. When you select Objective-C UnitKit class in the New File dialog box, the class created is already marked as conforming to this protocol.
2. Any method within a class conforming to the `UKTest` protocol that starts with the prefix "`test`" is considered to be a test method.
3. Each test method is executed on a freshly instantiated instance of the test class. Roughly in pseudo-code, UnitKit is doing the following:

```
foreach testMethod in testClass {
    id obj = [[testClass alloc] init];
    [obj testMethod];
    [obj release];
}
```

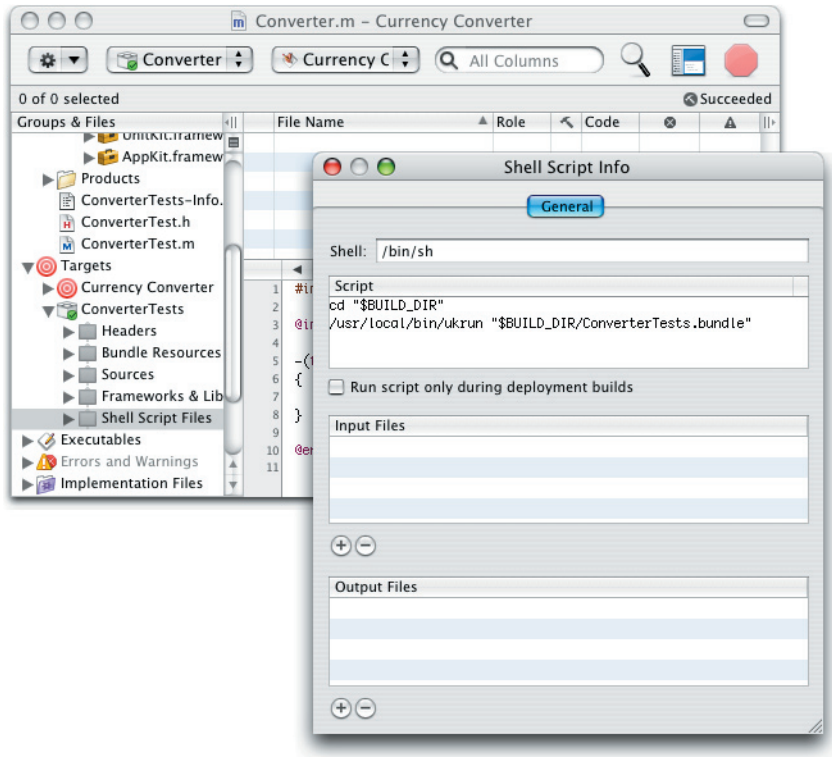
As of UnitKit 1.1, test classes and test methods are executed in alphabetical order. This makes it easier to find a particular test's output in the output from ukrun.

It's a little more complicated that this of course, but this is the general idea.

We're going to anticipate the question that is probably lurking on the tongue of anybody who has extensively used a JUnit based framework: What about `setUp` and `tearDown` methods? The answer is: Objective-C already provides this in the form of `init` and `dealloc` methods. Duplicating this functionality in another set of methods didn't seem quite right.

Figure 7

Taking a look at the shell script phase of the ConverterTest target. This is what executes ukrun which in turns runs all the tests in the bundle.



How Tests Bundles are Executed

You may be wondering what happens between the time you push the build button and when tests failures are reported in the detailed Build results window. Let's demystify that, shall we.

First, when you add a test bundle target to your project, a Shell Script Files phase is already part of the target. If you open up the target and then use the File > Get Info menu (⌘-I), you'll see that the last thing that the target does is execute `ukrun`. This is shown in Figure 7.

This is all there is to UnitKit's integration with Xcode. No other magic is involved. Any tool that wants to integrate in a similar fashion can use this same mechanism.

When `ukrun` executes, it opens up the given bundle and looks for all the test classes in it for execution. As it executes them, it writes out information to its standard out which is in turn interpreted by Xcode for display. This is the same output that is displayed in the Build window. Here's an example:

```
/Users/duncan/temp/Currency Converter/ConverterTest.m:21
Passed, expected 90.889999-90.910004, got 90.908997
/Users/duncan/temp/Currency Converter/ConverterTest.m:23:
warning: Failed
Result: 1 classes, 1 methods, 2 tests, 1 failed
```

The first line is the result of a passing test. The second is the result of a failing test. These lines are in same basic format as output by GCC and the other command line tools used by Xcode. By leveraging the format, UnitKit easily plugs in to Xcode. When Xcode sees a warning in this format, the warning and its details show up in the Build Results window.

How the Test Macros Work

So how does a statement like `UKIntsEqual(1, 2)` work? What happens to turn what you see into code into a warning statement in Xcode? Let's take a look.

First off, the `UKIntsEqual` macro is defined in UKTest.h as:

```
#define UKIntsEqual(a, b) [[UKTestHandler handler]
testInt:(a) equalTo:(b) inFile:__FILE__ line:__LINE__]
```

__FILE__ and __LINE__ are special preprocessor tokens that are replaced with the filename and line number of the source file that they are located in.

This means that when you compile a test class, the C preprocessor replaces the `UKIntsEqual(1, 2)` statement with the following:

```
[[UKTestHandler handler] testInt:1 equalTo:2
    inFile:"/Users/duncan/Project/Test.c" line:127]
```

This code sample is simplified from the actual logic in the UKTestHandler class for illustration purposes.

When run, this calls the `testInt:equalTo:inFile:line:` instance method of the UKTestHandler class. This method performs the following logic:

```
if (a == b) {
    [self reportStatus:YES inFile:file line:line
        message:msg];
} else {
    [self reportStatus:NO inFile:file line:line \
        message:msg];
}
```

This seems simple enough, right?

The `reportStatus:inFile:line:message:` method takes care of reporting the results of the test. By default, this method simply prints out a line to standard out indicating what happened.

The only exception to this is if a delegate is set on the test handler. In this case, the delegate will be sent a `reportStatus:inFile:line:message:` message and will be responsible for reporting the result of the test.

Providing Your Own Tests

If the compliment of built-in test macros isn't sufficient, you can provide your own tests. You'll probably want to follow the same pattern used by UnitKit (provide a macro which calls a test function), but the only requirement is that your test method calls the `reportStatus:inFile:line:message:` method of the default test handler.

Built-In Test Macros

UnitKit supports a fairly comprehensive list of test macros. Undoubtedly, more will be added in the future, but currently UnitKit supports the following set of tests.

UKPass()

The `UKPass` macro simply records a test passing. While running in Xcode, you typically won't see the effect of this macro unless you open up the detailed Build window and check the build transcript.

UKFail()

The `UKFail` macro records a test failure. When this macro is called, a test warning will be issued. You will see this test warning in Xcode's Build Results window.

UKTrue(condition)

The `UKTrue` macro evaluates the given condition. If the condition evaluates to true, the test will pass. If false, the test will fail and the following error message will be reported:

```
Failed, expected true, got false
```

Examples:

```
UKTrue(41 < 42);
UKTrue([@"/path" hasPrefix:@"/"]);
```

UKFalse(condition)

The `UKFalse` macro evaluates the given condition. If the condition evaluates to false, the test will pass. If true, the test will fail and the following error message will be reported:

```
Failed, expected false, got true
```

UKNil(ref)

The `UKNil` macro passes if the given reference is equal to `nil`. If the reference is not equal to nil, the test will fail and the following error message will be reported:

```
Failed, expected nil, got %@
```

UKNotNil(ref)

The `UKNotNil` macro passes if the given reference is not equal to nil. If the reference is equal to nil, the test will fail and the following error message will be reported:

```
Failed, expected not nil, got nil
```

UKIntsEqual(a, b)

The `UKIntsEqual` macro passes if the given int values equal each other. If the ints are not equal, the test will fail and the following error message will be reported:

```
Failed, expected %i, got %i
```

UKIntsNotEqual(a, b)

The `UKIntsNotEqual` macro passes if the given int values do not equal each other. If the ints are equal, the test will fail and the following error message will be reported:

```
Failed, didn't expect %i, got %i
```

UKFloatsEqual(a, b, delta)

The `UKFloatsEqual` macro passes if the given float values are equal to each other within the precision given by `delta`. If the float values are not equal, the test will fail and the following error message will be reported:

```
Failed, expected %f-%f, got %f
```

UKFloatsNotEqual(a, b, delta)

The `UKFloatsNotEqual` macros passes if the given floats are not equal to each other within the precision given by `delta`. If the float values are equal, the test will fail and the following error message will be reported:

```
Failed, didn't expect %f-%f, got %f
```

UKObjectsEqual(a, b)

The `UKObjectsEqual` macro passes if the objects are equal to each other according to the `isEqual:` method of object `a`. If the objects are not equal, the test will fail and the following error message will be reported:

```
Failed, expected %@, got %@
```

UKObjectsNotEqual(a, b)

The `UKObjectsNotEqual` macro passes if the objects are not equal to each other according to the `isEqual:` method of `a`. if the objects are not equal, the following message is reported:

```
Failed, didn't expect %@, got %@
```

UKObjectsSame(a, b)

The `UKObjectsSame` macro passes if the objects are exactly the same, i.e. they have the same pointer. If the objects are not the same, the following message is reported:

```
Failed, expected %@, got %@
```

UKObjectsNotSame(a, b)

The `UKObjectsNotSame` macro passes if the objects are not exactly the same, i.e. they don't have the same pointer. If the objects are equal, the following message is reported:

```
Failed, didn't expect %@, got %@
```

UKStringsEqual(a, b)

The `UKStringsEqual` macro passes if the strings are equal according to the `isEqualToString:` method of `a`. If the the strings are not equal, the following message is reported:

```
Failed, expected %@, got %@
```

UKStringsNotEqual(a, b)

The `UKStringsNotEqual` macro passes if the strings are not equal according to the `isEqualToString:` method of `a`. If the the strings are equal, the following message is reported:

```
Failed, didn't expect %@, got %@
```

UKStringContains(string, substring)

The `UKStringContains` macro passes if `string` contains `substring`. If `string` does not contain `substring`, the following message is reported:

```
Failed, %@ doesn't contain %@
```

UKStringDoesNotContain(string, substring)

The `UKStringDoesNotContain` macro passes if `string` does not contain `substring`. If `string` does contain `substring` the following message is reported:

```
Failed, %@ contains %@
```

UKRaisesException(expression)

The `UKRaisesException` macro passes if `expression` raises an execption when it is executed. If an exception is not raised, the following message is reported:

```
Failed, an exception was not raised
```

UKDoesNotRaiseException(expression)

The `UKDoesNotRaiseException` macro passes if `expression` does not raise an exception when it is executed. If an exception is raised, the following is reported:

```
Failed, an exeception %@ was raised
```

UKRaisesExceptionNamed(expression, name)

The `UKRaisesExceptionNamed` macro passes if `expression` raises an exception with the given name when it is executed.

UKRaisesExceptionClass(expression, class)

The `UKRaisesExceptionClass` macro passes if `expression` raises an exception with the given class when it is executed.

Creating Your Own Tests

UnitKit ships with a fairly comprehensive set of tests but your needs may be more specific. Creating your own test macros is fairly straightforward

No test kit can anticipate all needs. Occasionally, you might need to create your own assertions. We'll show you how. But not yet. Not in this version of the manual. Patience.

Running Test Code in the Debugger

Sometimes the output from a Unit Kit test run isn't enough to see what is happening. When this happens, the debugger is the best tool for the job. However, since Unit Kit tests are run during the build process (so that the reporting of test failures show up as build errors), running tests in the debugger isn't immediately straightforward.

Here's how to do it:

1. Add a new custom executable to your project and give it a name such as "ukrun". Fill in the executable path with the following:

   ```
   /usr/local/bin/ukrun
   ```

2. Select the custom executable and bring then use the File > Get Info menu (⌘-I). Add an argument for each test bundle to run.

3. Set breakpoints in your test code, as you normally would.

4. Make sure the test bundle that contains the code is the current target and that ukrun is the current executable. You can set the current executable either by selecting the "Executables" group or by using the pull down on the Run Log window.

5. Build your code and then run the debugger. Execution will stop at your breakpoints.

Tips and Tricks

The following are some tips and tricks to help maximize your UnitKit pleasure:

- If you want to execute your unit tests each time you build your primary target—along with having Xcode refuse to build your primary target if the tests don't pass—make your primary target depend on the test bundle target. This enforces that the tests always pass. To do this, select the primary target and use the File > Get Info menu (⌘-I). In the Target Info window's General tab, click the + (Plus) button

under the Direct Dependencies list and add the test bundle target.

- If you see a the following warning message:

  ```
  Undefined symbols: .objc_class_name_UKTestHandler
  ```

  You'll need to make sure that the UnitKit framework is selected to be part of the test bundle build.

- If the code you are testing relies on embedded frameworks, either locate those frameworks in the build directory or edit the `FRAMEWORK_SEARCH_PATHS` environment variable to point at the directory (or directories) that contain the embedded frameworks. To edit this environment variable, get the Info window for the Test Bundle target, select the Build tab, and edit the Framework Search Paths setting.

UnitKit Design Notes

Building a unit test framework is actually pretty interesting and is full of design choices. Here are a collection of some of the design choices made in UnitKit:

Xcode Integration

This integration was finally realized in full form in Xcode 1.5. In previous versions of Xcode, it was necessary to open the Detailed Build Results window.

From the moment that Xcode was first demonstrated at WWDC2003, it was obvious that the new Errors & Warnings group would be perfect for reporting unit test errors. UnitKit's design philosophy revolves around being run from Xcode and reporting test results in a fashion that Xcode can present to the user.

At this point, Xcode integration is pretty simple. Test results are reported out to stdout in the form of filename:linenumber: [error|warning]:... Xcode picks these errors up and reports them as you would expect. Because of this tight integration into the Xcode workflow, everything that's superfluous to this has been ejected. Simple is better. UnitKit is purpose built and makes no apologies.

Separate Bundle Targets for Testing

Some test frameworks mandate that you mix tests in with the finished executable. Others set things up so that your tests remain separate. UnitKit keeps the tests separate for one primary reason: according to Apple engineers, the performance of an ap-

plication in standard use by end users is greatly affected by the size of your executable. Shipping your tests as part of the core executable increases its size and has a corresponding effect on the performance of your app.

This design goal however isn't set in stone. Investigations are ongoing as to how to leverage Zero-Link or the Xcode build system so that tests can be built as part of a development executable, but excluded from a distribution build. Of course, if this does come to pass, UnitKit will always be compatible with the current bundle target test setup.

Simple Core Design

Unit test frameworks have always seemed to be more complicated than really needed. UnitKit strives to be *exactly* as complicated as needed. To that end, there are only three major parts to UnitKit:

- The test macros in `UKTest.h`
- Test execution and reporting, as implemented in the UKTestHandler class
- Test detection as implemented in the UKRunner class

Over time, this core design will get a bit more complicated. But attempts to bloat will be resisted.

Using C Preprocessor Macros for Test Assertions

This one really seems to have an affect on those steeped heavily in the JUnit tradition. UnitKit uses C preprocessor macros to define its test assertions. This makes sense for two primary reasons:

- Macros let UnitKit grab the filename and line number location of a test assertion at compile time. If you look into the macros themselves, you'll notice that they use __LINE__ and __FILE__ to report this information.
- It reduces the number of classes in the system. Sure, objects and classes are good things. But when all you need to do is report an assertion to the test machinery, a macro works great.

Macros aren't without their share of pain. Compilation errors within a macro can be problematic to chase down. This is a

good reason not to use them casually. UnitKit has been carefully designed so that a minimum of logic is contained within the macro. Essentially each test macro make a quick method call out to UKTestHandler.

Internationalization (i18n)

The small amount of UnitKit UI that peeks out from behind Xcode is composed of the strings that pop up in the Errors & Warnings window in the form of status messages. These messages are retrieved from the UKTestHandler.strings file and are easily localizable into other languages.

Using a Singleton Test Handler (Snake eats Tail)

Turns out that using UnitKit to test itself was a bit complicated. For that matter, any test kit testing itself has to cope with one primary issue: How to report test failures to itself without reporting them to the user.

UnitKit solves this by using a singleton UKTestHandler that can accept a delegate. When the delegate is not set, UKTestHandler reports errors to stdout allowing Xcode to do its thing. But when a delegate is set, it gets notifications of all the tests as they run. This allows UnitKit to test itself within itself and still remain simple.

Objective-C Runtime Fun

The UKRunner class finds and starts the execution of all test classes (those that conform to the UKTest protocol) in a bundle. Problem is that there's not a nifty method on NSBundle to ask it what classes it brought into the party. But you can find out by diving into the Objective-C runtime and getting a list of all of the classes in the system, then going through that list and finding out which bundle they were loaded from. It's not the most efficient method, but so far hasn't produced a net performance loss.

This functionality is in the UKTestClassesFromBundle(bundle) function located in UKRunner.m. It's not too gross, but if there's a better way to do this, please suggest away!

Feedback

Comments, questions, rants, and raves are always welcome. Please send them to:

```
duncan@x180.net
```

As well, there is a mailing list set up for UnitKit users to mingle and share ideas as well as get news about UnitKit developments. To subscribe to this mailing list, send email to:

```
unitkit-subscribe@unitkit.org
```

You can also check the website for more details about the mailing list.

Acknowledgements

Many people have given ideas, suggestions, and feedback to UnitKit. In particular Mike Clark, Joseph Heck, Glenn Vanderburg, and Daniel Steinberg provided feedback during the early development of UnitKit that materially shaped what you see now.

Many more people have contributed to UnitKit since its introduction. You can find a list of contributors in the `NOTICE` file that is located in the UnitKit distribution.

Thanks also to the attendees of the UnitKit session at AdHoc/MacHack 19 who asked some great questions and gave great feedback about the test framework. If you've never been to AdHoc/MacHack, well, you just need to go next time.