Mobile Application Testing

Adrian Kosmaczewski

Contents

List of Figures

List of Tables

About the author

Adrian Kosmaczewski is a software developer, trainer and book author. He has shipped mobile, web and desktop apps for iOS, Android, Mac OS X, Windows and Linux since 1996.

Adrian is the author of "Mobile JavaScript Application Development" and "Sencha Touch 2: Up and Running", both published by O'Reilly.

When not writing or teaching, Adrian likes to spend time with his wife Claudia and his cat Max. He updates his blogs and his Twitter or Instagram accounts ("@akosma") as often as possible, and is happy to have new followers every day.

Adrian has studied physics in Switzerland, economics in Buenos Aires, and holds a Master in Information Technology with a specialization in Software Engineering from the University of Liverpool.

Preface

Mobile applications have taken the world by storm. Thousands of new applications are published every day, and they are getting every day more complex as customer demand more and more from their apps.

The need for high quality standards in mobile apps will only increase in the future; this booklet is a humble compilation of tips, tricks and techniques that can yield better code and happier teams.

I hope that these lines will help you to increase your productivity and the sales of your apps!

Oron-la-Ville, January 2013.

1 Introduction

Mostly driven by iOS and Android, mobile developers are faced with increasingly complex requirements and deployment issues, as the market for mobile applications grows constantly.

This book is a practical guide for testing mobile applications for iOS or Android devices. It includes an overview of the most important tools available, either provided from the platform vendors, or available in the market as open source or commercially.

1.1 Audience of this Book

This book is aimed to mobile developers, proficient in either iOS and/or Android, willing to increase the quality of their applications. They should be comfortable with the general concepts of software quality management, which will not be covered in this book.

1.2 Structure of the Book

This book is divided in two sections, covering the same subjects for both iOS and Android applications:

- Defensive coding techniques;
- Unit testing;
- Functional testing.

For each platform, the book covers the most important frameworks and tools available in each area.

Sample Application

We are going to use a very simple application to showcase different technigues used to test iOS and Android apps: an integer calculator, implemented in the most naive and simple way.

This application uses an MVC architecture, using a class named `AKOIntegerCalculator` for iOS and `IntegerCalculator` for Android, encapsulating the calculation logic, and an `AKOViewController` class (respectively, `MainActivity` for Android) to drive the user interface. Figure 1.1 and Figure 1.2 show the UI of the application running in both the iOS Simulator and the Android Emulator, which offers basic operations and will be tested throughout the following chapters.

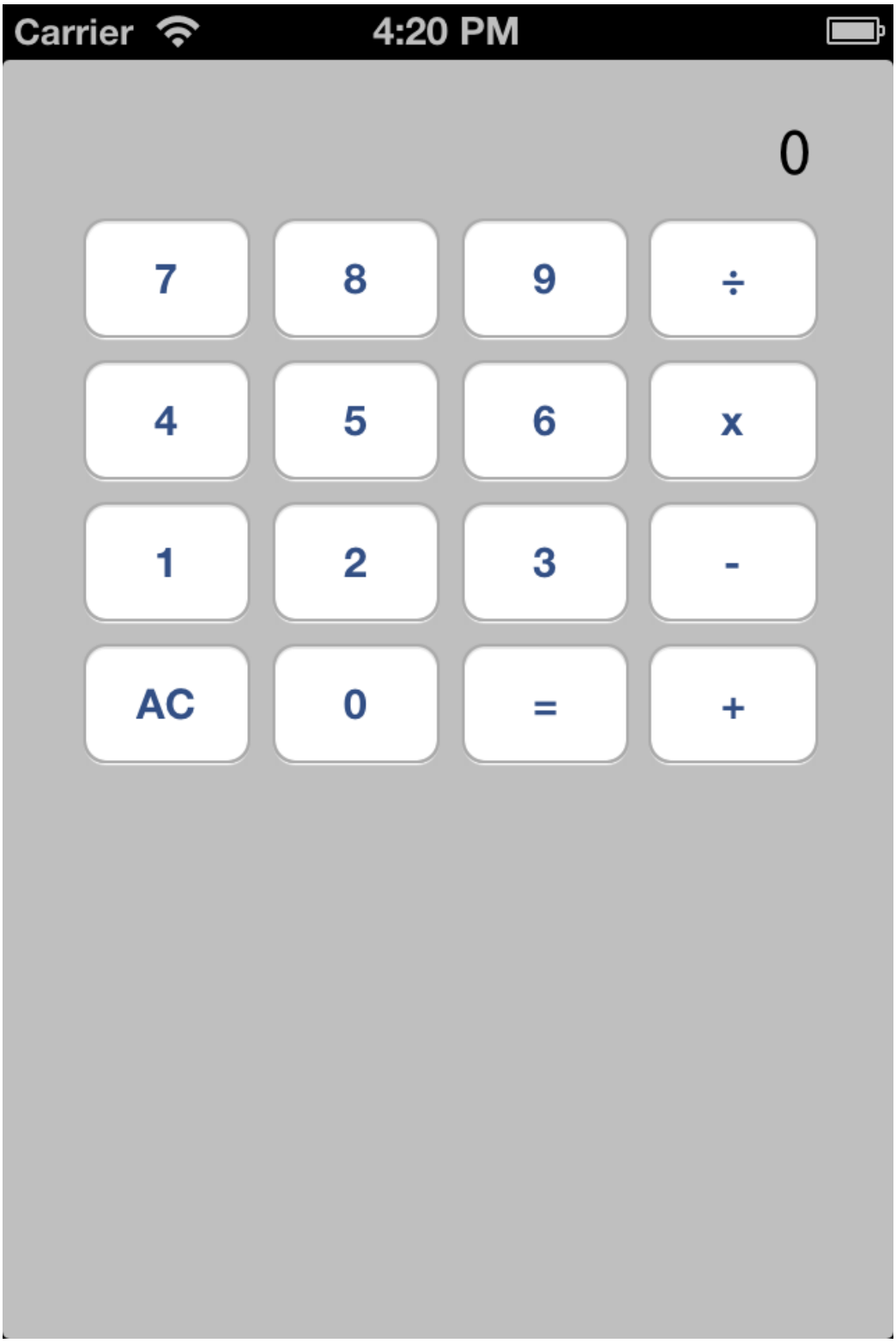

Figure 1.1: iOS sample application

Figure 1.2: Android sample application

1.3 Technical Requirements

To use the code examples and the instructions of this book, you should use a computer with the following specifications:

- Mac with OS X Mountain Lion and the latest Xcode (available for free in the Mac App Store), and the latest Android SDK already installed, including Eclipse.
- Windows or Linux laptop with the latest Android SDK and Eclipse.
- iOS and/or Android devices (provisioned and ready to be used in development.)

In all cases, the computers used during this training should include:

- A working local web server, ideally available through the local network, like the one provided by MAMP.
- A working Ruby 1.9 installation, including RubyGems.
- Git 1.8 or later.
- (optional but recommended on Macs) Homebrew, a package manager for OS X.

Note

At the time of this writing, the latest version of Xcode is 4.6, and the latest version of the Android developer tools is version 21.0.1, which is composed by the following components:

- Eclipse 3.8.1 (Juno)
- Android 4.2 (API 17)

Readers of this book must be comfortable using a terminal or a command line.

1.4 Acknowledgements

The author would like to thank Duncan Scholtz, Java developer and Kishyr Ramdial, iOS developer at immedia (Durban, South Africa) for his help and precious insight during the development of this booklet.

Part I

Testing iOS Applications

2 Defensive Coding Techniques for iOS

This chapter will provide a short introduction to techniques used to increase the quality of your iOS applications.

2.1 NSError, NSException and NSAssert

Cocoa offers different mechanisms to notify about unexpected critical events. They are meant to be used in different contexts, ranging from coding time to runtime.

NSAssert

An assertion is code that's used during development that allows a program to check itself as it runs. When an assertion is true, that means that everything is operating as expected. When it's false, that means that it has detected an unexpected error in the code. They are specially useful in large, complicated programs and in high-reliability systems.

When to use Assertions

Assertions can be used to check the following assumptions:

- That the values of function or method parameters fall within expected ranges;
- That files are open (or closed) when a routine starts (or ends) executing;
- That files are writeable (or not);
- That the value of an input is not changed by a function;
- That a pointer is not null;

- That an array contains at least a certain number of elements;
- That an array has been initialized to some state before execution;
- That a container is empty;
- That a computation result match a particular set of assumptions before being returned to the caller.

Guidelines for Assertions

Remember to follow the following guidelines when using assertions:

- Use error-handling for conditions that are expected to occur; use assertions for conditions that should never occur;
- Avoid putting executable code into assertions;
- Use assertions to check for pre- and post-conditions in methods and functions.
- For higher robustness, assert and then handle the error anyway.

Note

You can think of assertions as executable documentation, acting as meaningful comments helping developers understand the assumptions made by the creator of a particular set of code lines.

Assertions in Cocoa

Cocoa offers the following assertion macros to be used exclusively in Objective-C code:

```
Line 1  NSAssert(condition, desc, ...)
Line 2  NSAssert1(condition, desc, arg)
    -   NSAssert2(condition, desc, arg, arg)
    -   NSAssert3(condition, desc, arg, arg, arg)
    5   NSAssert4(condition, desc, arg, arg, arg, arg)
    -   NSAssert5(condition, desc, arg, arg, arg, arg, arg)
    -   NSParameterAssert(condition)
```

The same assertions are available for use inside of C functions:

```
Line 1  NSCAssert(condition, NSString *description, ...)
Line 2  NSCAssert1(condition, NSString *description, arg)
    -   NSCAssert2(condition, NSString *description, arg, arg)
    -   NSCAssert3(condition, NSString *description, arg, arg, arg)
    5   NSCAssert4(condition, NSString *description, arg, arg, arg, arg)
```

```
- NSCAssert5(condition, NSString *description, arg, arg, arg, arg, arg)
- NSCParameterAssert(condition)
```

Note

The `NSParameterAssert` and `NSCParameterAssert` macros can be used to validate a parameter of (respectively) an Objective-C method or C function. Just providing the parameter as the condition argument will trigger the evaluation, and if this yields a false value, the macro will log an error message and raise an exception.

An `NSAssert` will throw an exception when it fails. So `NSAssert` is there to be short and easy way to write and to check any assumptions you have made in your code. It is not an alternative to exceptions, just a shortcut. If an assertion fails then something has gone terribly wrong in your code and the program should not continue.

One thing to note is that `NSAssert` will not be compiled into your code in a release build.

NSException

Exceptions, as the name implies, are used to signal exceptional events that threaten the correct execution of an application, and are most probably due to programming errors or runtime conditions out of reach from the developer.

The times you would `@throw` your own `NSException` are when you definitely want it in a release build, and in things like public libraries/interface when some arguments are invalid or you have been called incorrectly. Note that it isn't really standard practice to `@catch` an exception and continue running your application. If you try this with some of Apple's standard libraries (for example Core Data) bad things can happen. Similar to an assert, if an exception is thrown the app should generally terminate fairly quickly because it means there is a programming error somewhere.

Guidelines for Using Exceptions

Follow these guidelines when using exceptions in your applications:

- Use exceptions to notify about errors that cannot be ignored.
- Throw only when exceptional situations occur.
- Do not throw if you can solve the error locally.

- Avoid throwing in `init` or `dealloc` (to avoid memory leaks and inconsistencies) unless you `@catch` inside of them.
- Think of exceptions as part of the class interface; clients should be aware of them, in a similar way to Java's own checked exceptions.
- Include in the exception message all the information that led to the exception.
- Avoid empty `@catch` blocks.
- Consider having a centralized exception reporter.
- Standardize the use of exceptions in your projects.
- Consider alternatives (logging, NSError, etc).

Setting a Global Uncaught Exception Handler

Cocoa allows iOS developers to set a global uncaught exception handler to their applications:

```
Line 1  void exceptionHandler(NSException *exception)
-       {
-           [[NSUserDefaults standardUserDefaults] setBool:YES forKey: ↩
                DID_CRASH];
-           [[NSUserDefaults standardUserDefaults] synchronize];
5       }
-
-       @implementation AKOAppDelegate
-
-       - (BOOL)application:(UIApplication *)application  ↩
            didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
10      {
-           NSSetUncaughtExceptionHandler(&exceptionHandler);
-
-           BOOL didCrash = [[NSUserDefaults standardUserDefaults] boolForKey ↩
                :DID_CRASH];
-           if (didCrash)
15          {
-               [[NSUserDefaults standardUserDefaults] setBool:NO forKey: ↩
                    DID_CRASH];
-               NSString *message = @"Unfortunately this app crashed last  ↩
                    time it was run... Please contact the developer and ask  ↩
                    for a refund.";
Line 18         UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@" ↩
                    Crash!"
```

```
                                                        message: ↩
                                                            message
                                                       delegate:nil
                                                cancelButtonTitle:@"OK"
                                                otherButtonTitles:nil];
        [alert show];
    }

    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen  ↩
        mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[AKOViewController alloc] initWithNibName: ↩
        @"AKOViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

@end
```

Although the application dies shortly after executing the global uncaught exception handler, a common technique is to set a `applicationDidCrash = YES` value in `NSUserDefaults` and to check that value when the application starts again.

NSError

`NSErrors` should be used in your libraries/interfaces for errors that are not programming errors, and that can be recovered from. You can provide information/error codes to the caller and they can handle the error cleanly, alert the user if appropriate, and continue execution. This would typically be for things like a "File not found" error or some other non-fatal error.

Note

To put it more strongly, an NSException should not be used to indicate a recoverable error. In other words: Obj-C's NSException == Java's Error class, and Obj-C's NSError == Java's Exception class.

Error Handling Techniques

To handle errors, a certain routine might choose to do any (or all) of the following:

- Return neutral values, such as `0`, `@""` (empty string) or a `NULL` pointer.

- Skip to the next piece of valid data, such as when dealing with long lists of records sequentially.
- Return the same answer as the last time the method was called.
- Substitute the return value by the closest leval value.
- Log the event in a log file.
- Return an error code (very common in Cocoa, using an `NSError **` parameter).
- Call an error processing routine.
- Display an error message whenever the error appears (beware of security and usability implications!)
- Shut down (for critical systems, it might be the only possible choice!)

2.2 Code Defensively

This section showcases some simple techniques that you might want to adapt to your own team workflow. The objective being that the code produced by your organization is predictable and consistent.

Treat warnings as errors

First of all, why does the Objective-C compiler (or compilers in general) output "warnings"? Many developers are puzzled the first time they encounter them, since even if the compiler complained, the application usually runs anyway without (perceptible) problems.

Warnings are used to signal specific issues in the source code which could potentially lead to crashes or misbehavior under some circumstances, but which should not (pay attention to the verb "should") block the normal compilation and (hopefully) execution of your code (otherwise, it would be a compiler error).

It's the way used by your compiler to say:

> Hey, I'm not sure, but there's something fishy in here.

Not removing warnings, as I said above, is a problem that originates both in the programming background of the developer, and specific technical issues.

Culturally speaking, many other programming environments either do not have compilers at all (at least not "visible" ones, like Ruby or PHP) or simply do not spit warnings for anything else than deprecated methods (like C# or Java); this

situation has made many developers new to the iPhone platform to blatantly ignore them.

Technically, given the fact that Objective-C is the "other" object-oriented superset of C, and that it behaves as a coin with both a static and a dynamic side, compiler warnings convey a great amount of precious information that must **never** be ignored.

In this sense, Objective-C has a lot in common with C. `Ignoring warnings in C` is strongly discouraged, and Scott Meyers explains this in chapter 9 of his book "Effective C++", stating that (third edition, page 263):

> Take compiler warnings seriously, and strive to compile warning-free at the maximum warning level supported by your compilers.

In the case of Objective-C, this can be done by setting a particular value named GCC_TREAT_WARNINGS_AS_ERRORS (-Werror) to true in your build settings, as shown in the figure below.

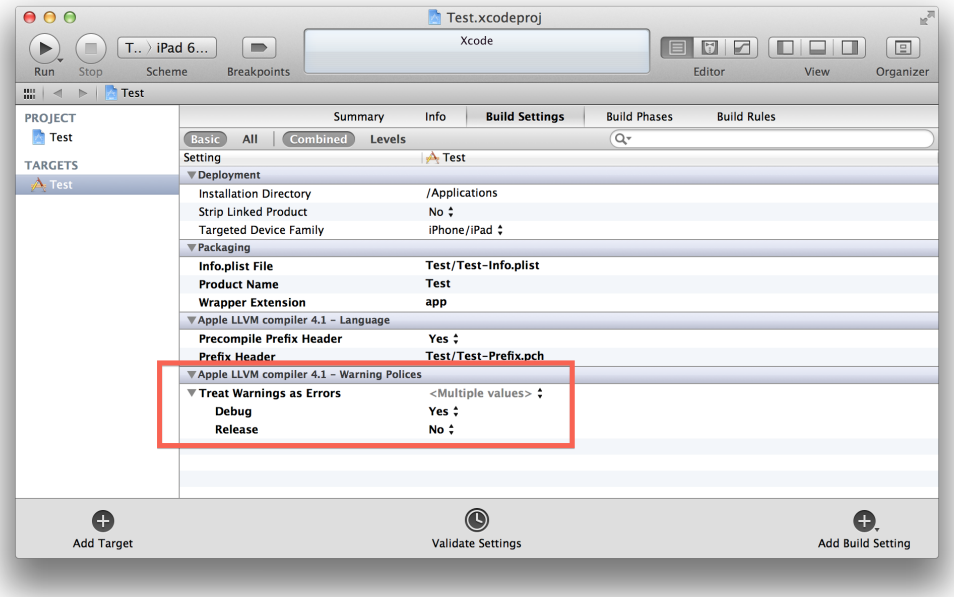

Figure 2.1: Treating warnings as errors in Xcode 4

Steve McConnell takes this advice to another level of importance in his classic book "Code Complete" (second edition, page 557):

> Set your compiler's warning level to the highest, pickiest level possible, and fix the errors it reports. It's sloppy to ignore compiler errors. It's

> even sloppier to turn off the warnings so that you can't even see them. Children sometimes think that if they close their eyes and can't see you, they've made you go away (. . .).
>
> Assume that the people who wrote the compiler know a great deal more about your language than you do. If they're warning you about something, it usually means you have an opportunity to learn something new about your language.

To give a concrete example of the importance of warnings, many of us have had to migrate applications developed for iPhone OS 2.x to the 3.0 operating system, mostly because failure to run on the new version of the OS was ground for removal from the App Store. That moment of truth, the rebuild of the Xcode project, unveiled a plethora of compiler warnings, most due to deprecated methods, like the tableView:accessoryTypeForRowWithIndexPath: method of the UITableViewDelegate protocol, or the initWithFrame:reuseIdentifier: method of the UITableViewCell class (which, incidentally, are properly marked as such in the documentation, too).

Compiler warnings in Objective-C have a multitude of reasons:

- Using deprecated symbols;
- Calling method names not declared in included headers;
- Calling methods belonging to implicit protocols;
- Using some ambiguous commands which might be intentional but are syntactically valid anyway;
- Forgetting to return a result in methods not returning "void";
- Forgetting to #import the header file of a class declared as a forward "@class";
- Downcasting values and pointers implicitly.

Organize your code

Each @implementation *.m file should always present methods in this order:

1. init and dealloc
2. public methods
3. public @dynamic properties
4. delegate methods (for each supported protocol)
5. private methods

Use #pragma mark statements

Each logic group of methods should be separated from each other using the following lines (just type "#p" and hit the TAB key in Xcode to speed up the process):

```
Line 1  // ...
    -       return cell;
    -   }
    -
    5   #pragma mark - UIAlertViewDelegate methods
    -
    -   - (void)alertView:(UIAlertView *)alertView
    -           clickedButtonAtIndex:(NSInteger)buttonIndex
    -   {
   10   // ...
```

The advantage of this approach is that later, you can use those #pragma marks to generate an automatic layout in the symbols pop-up of Xcode, as shown in the figure below.

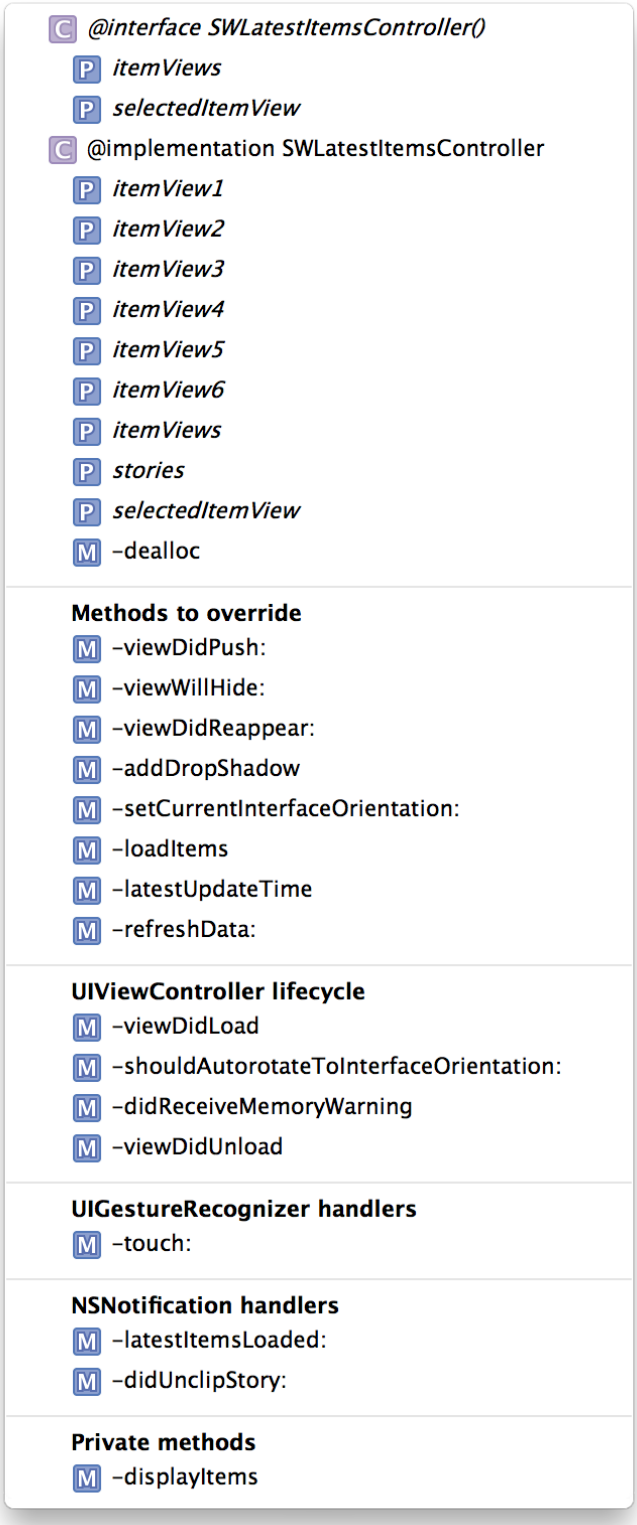

Figure 2.2: Code regions as seen in the Xcode Jump Bar

You can get this pop-up window clicking on the “Jump Bar” of the current Xcode editor.

Only advertise public methods in header files

This means putting private methods definitions in a (Private) category on top of the *.m file. This will remove all compiler warnings (about “this class might not respond to this selector”) and will cleanly separate what’s public from what’s not.

```
Line 1  #import "UntitledViewController.h"
-
-       @interface UntitledViewController ()
-       - (id)returnPrivateObject;
5       - (void)changeInternalState:(NSString *)param;
-       @end
-
-       @implementation UntitledViewController
-
10      - (id)init
-       {
```

Use the Scientific Method of Debugging

describes a scientific method for debugging sessions, which is useful to remind in this context:

1. Stabilize the error
2. Locate the source of the error:
   a. Gather the data that produces the defect.
   b. Analyze the data that has been gathered, and form a hypothesis about the defect.
   c. Determine how to prove or disprove the hypothesis, either by testing the program or by examining the code.
   d. Prove or disprove the hypothesis by using the procedure identified in 2(c).
3. Fix the defect.
4. Test the fix.
5. Look for similar errors.

Use consistent coding conventions

At the end of this document, Appendix A contains generic and important coding conventions. You can start from this document to create your own.

2.3 Debugging Techniques

This section will highlight interesting debugging techniques available in iOS.

Add Context Information to Log Messages

The C preprocessor provides a number of standard macros that give you information about the current file, line number, or function. Additionally, Objective-C has the _cmd implicit argument which gives the selector of the current method, and functions for converting selectors and classes to strings. You can use these in your NSLog statements to provide useful context during debugging or error handling.

```
Line 1  NSMutableArray *someObject = [NSMutableArray array];
    -   NSLog(@"%s:%d someObject=%@", __func__, __LINE__, someObject);
    -   [someObject addObject:@"foo"];
    -   NSLog(@"%s:%d someObject=%@", __func__, __LINE__, someObject);
```

Table 2.1: Preprocessor macros and for logging in C/C++/Objective-C.

Macro	**Format Specifier**	**Description**
__func__	%s	Current function signature.
__LINE__	%d	Current line number in the source code file
__FILE__	%s	Full path to the source code file.
__PRETTY_FUNCTION__	%s	Like *func*, but includes verbose type information in C++ code.

Table 2.2: Expressions for logging in Objective-C.

Expression	Format Specifier	Description
`NSStringFromSelector(_cmd)`	%@	Name of the current selector.
`NSStringFromClass([self class])`	%@	Name of the current object's class.
`[[NSString stringWithUTF8String:__FILE__] lastPathComponent]`	%@	Name of the source code file.
`[NSThread callStackSymbols]`	%@	NSArray of the current stack trace as programmer-readable strings. For debugging only, do not present it to end users or use to do any logic in your program.

Inspecting Objects

All Cocoa objects (everything derived from `NSObject`) support a `description` method that returns an `NSString` describing the object. The most convenient way to access this description is via Xcode's Print Description to Console menu command. Alternatively, you can use LLDB's `print-object` (or `po` for short) command.

Adding Exception Breakpoints

Xcode 4 offers a simple way to generate a breakpoint whenever an exception occurs. A great idea when you need to track down application crashes.

In the break point navigator click the plus (+) symbol in the lower left corner. Choose "Add Exception Breakpoint". The exception breakpoint options are shown in Figure 2.3.

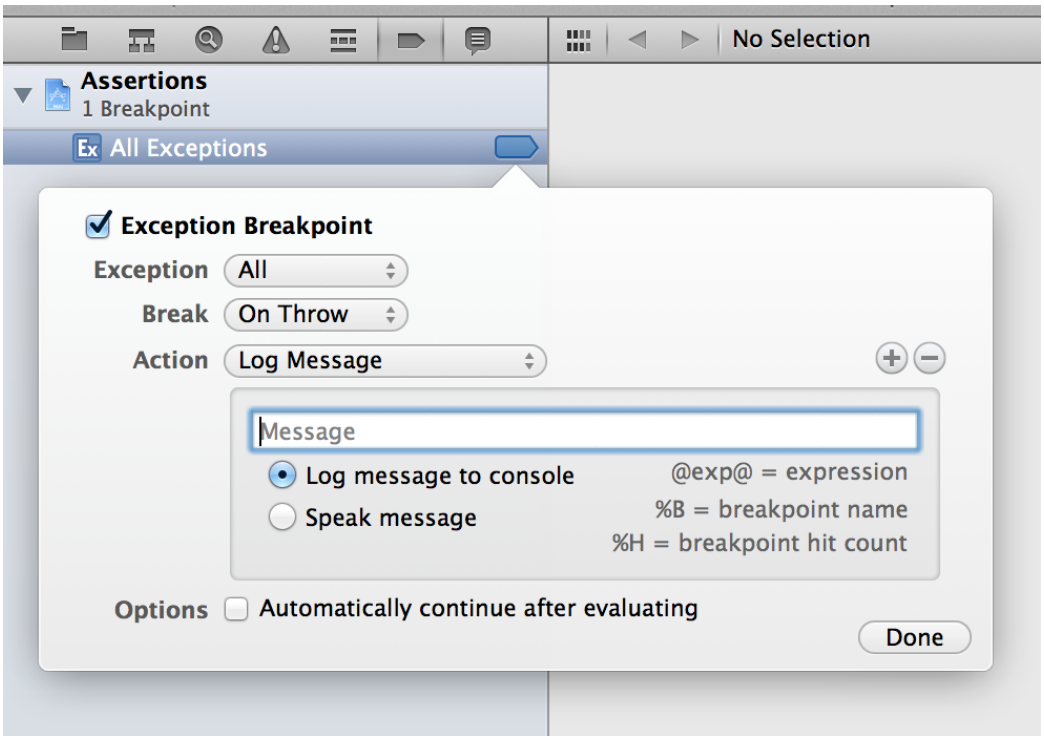

Figure 2.3: Setting an exception breakpoint

Beware though that all exceptions will be logged by this breakpoint, not only uncaught exceptions, but also those surrounded by `@try` / `@catch` statements. You might want to check the "Automatically Continue after Evaluating" checkbox to avoid stopping too often.

Inspecting Memory Management

You can use `-retainCount` to get the current retain count of an object. While this can sometimes be a useful debugging aid, be very careful when you interpret the results. For example:

```
Line 1  (lldb) set $s=(void *)[NSClassFromString(@"NSString") string]
    -   (lldb) p (int)[$s retainCount]
    -   $4 = 2147483647
    -   (lldb) p/x 2147483647
    5   $5 = 0x7fffffff
```

The system maintains a set of singleton strings for commonly used values, like the empty string. The retain count for these strings is a special value indicating that the object can't be released.

Another common source of confusion is the autorelease mechanism. If an object has been autoreleased, its retain count is higher than you might otherwise think, a fact that's compensated for by the autorelease pool releasing it at some point in the future. You can determine what objects are in what autorelease pools by calling `_CFAutoreleasePoolPrintPools` to print the contents of all the autorelease pools on the autorelease pool stack:

```
(lldb) call (void)_CFAutoreleasePoolPrintPools()
- -- ---- -------- Autorelease Pools -------- ---- -- -
==== top of stack ================
  0x327890 (NSCFDictionary)
  0x32cf30 (NSCFNumber)
  [...]
==== top of pool, 10 objects ===============
  0x306160 (__NSArray0)
  0x127020 (NSEvent)
  0x127f60 (NSEvent)
==== top of pool, 3 objects ================
- -- ---- -------- ----------------- -------- ---- -- -
```

Zombies

A common type of bug when programming with Cocoa is over-releasing an object. This typically causes your application to crash, but the crash occurs after the last reference count is released (when you try to message the freed object), which is usually quite removed from the original bug. `NSZombieEnabled` is your best bet for debugging this sort of problems; it will uncover any attempt to interact with a freed object.

The easiest way to enable zombies is via Instruments, as shown in Figure 2.4. However, you can also enable zombies in Xcode, as shown in Figure 2.5. To do that, hit the "Option-Command-R" key combination (or select "Product / Run" while holding the "Option" key) and select the "Diagnostics" tab; check the "Enable Zombie Objects" option in the dialog.

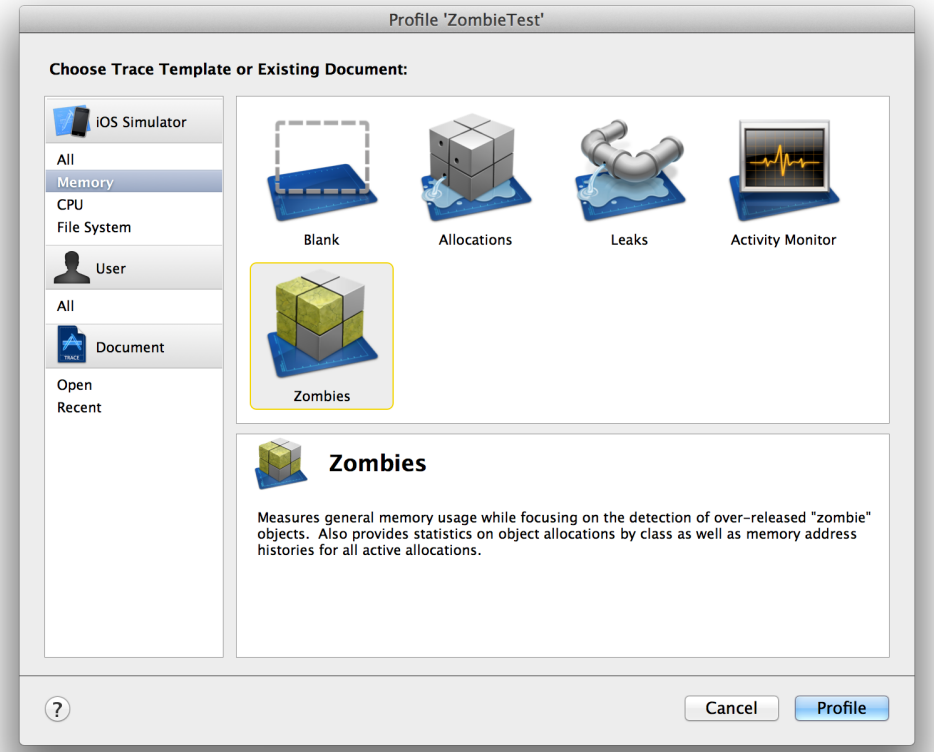

Figure 2.4: Using the zombies instruments

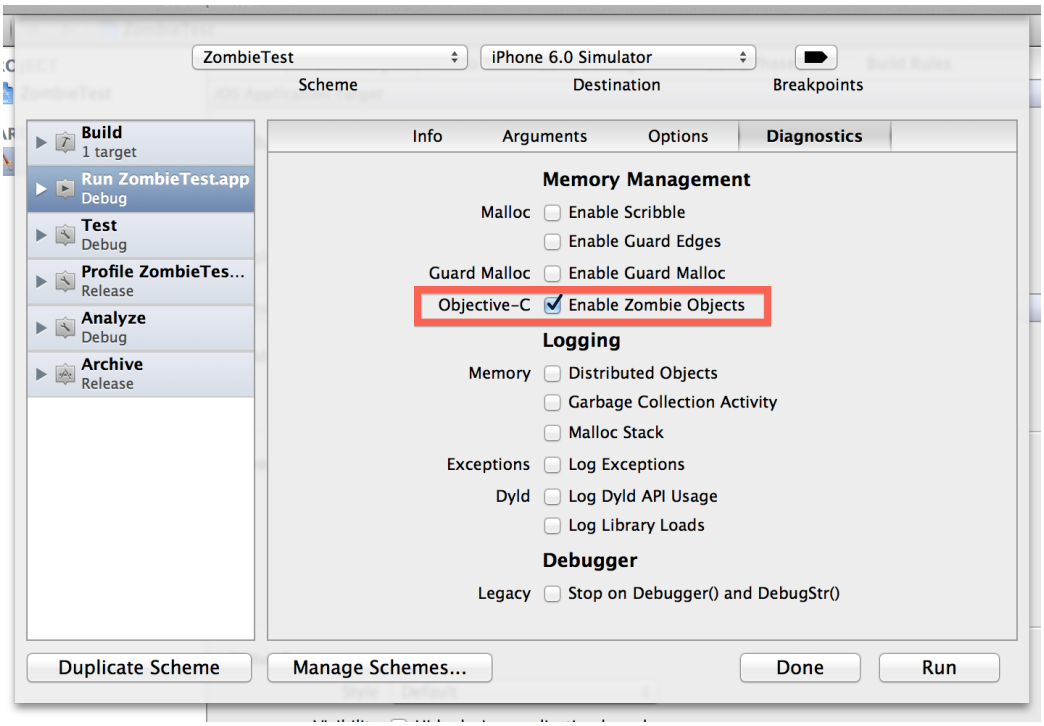

Figure 2.5: Enabling zombies in Xcode 4.5

Key-Value Observing

If you're using Key-Value Observing and you want to know who is observing what on a particular object, you can get the observation information for that object and print it using using LLDB's `print-object` command.

```
(lldb) # self is some Objective-C object.
(lldb) po self
<ZoneInfoManager: 0x48340d0>
(lldb) # Let's see who's observing what key paths.
(lldb) po [self observationInfo]
<NSKeyValueObservationInfo 0x48702d0> (
<NSKeyValueObservance 0x4825490: Observer: 0x48436)
```

Finding Non-Localized Strings

You can set the `NSShowNonLocalizedStrings` preference to find strings that should have been localized but weren't. Once enabled, if you request a localized string and the string is not found in a strings file, the system will return the

string capitalized and log a message to the console. This is a great way to uncover problems with out-of-date localizations.

Debugging UIViews

`UIView` implements a useful `description` method. In addition, it implements a `recursiveDescription` method that you can call to get a summary of an entire view hierarchy.

```
(lldb) po [self view]
<UIView: 0x6a107c0; frame = (0 20; 320 460); autoresize = W+H; layer = [...]
Current language: auto; currently objective-c
(lldb) po [[self view] recursiveDescription]
<UIView: 0x6a107c0; frame = (0 20; 320 460); autoresize = W+H; layer = [...]
   | <UIRoundedRectButton: 0x6a103e0; frame = (124 196; 72 37); opaque = NO; [...]
      |    | <UIButtonLabel: 0x6a117b0; frame = (19 8; 34 21); text = 'Test'; [...]
```

Debugging Core Data Objects

If you're working on an app that uses Core Data, it's inevitable that you'll end up in the debugger and need to dig around in the object graph. You'll also quickly realize that Core Data's `-description` of an object isn't terribly helpful:

```
(lldb) po myListObj
(List *) $19 = 0x08054ac0 <List: 0x8054ac0> (entity: List; id: 0 x8074280 <x-coredata://29B10357-0723-4950-9EB6-E6D7AD6269B9/List/p175> ; data: <fault>)
```

Core Data's documentation is excellent, but surprisingly doesn't cover some of the tricks you can use to examine managed objects in the debugger.

The first trick is to fire a fault on the object using `-willAccessValueForKey`. After that, you can see what's really there:

```
(lldb) po [myListObj willAccessValueForKey:nil]
(id) $20 = 0x08054ac0 <List: 0x8054ac0> (entity: List; id: 0x8074280 <x-coredata://29B10357-0723-4950-9EB6-E6D7AD6269B9/List/p175> ; data: {
    containerId = "8E4652D3-5516-4186-B1C9-DDBE41E108CF";
    createdAt = "2009-10-08 15:17:53 +0000";
```

```
    itemContainers = "<relationship fault: 0x8020850 'itemContainers ↩
        '>";
})
```

You might also be surprised when you try to access one of the properties of the object:

```
Line 1  (lldb) po myListObj.containerId
-       error: property 'containerId' not found on object of type 'List *'
-       error: 1 errors parsing expression
```

Remember that these properties are defined as `@dynamic` and there's a lot of work done by Core Data at runtime to provide the implementation. The solution here is to use the KVC accessor `-valueForKey:` to get the object's value:

```
Line 1  (lldb) po [myListObj valueForKey:@"containerId"]
-       (id) $26 = 0x08069b60 8E4652D3-5516-4186-B1C9-DDBE41E108CF
```

Often, you'll want to examine the relationships between objects. As you can see in the output above, the attribute `itemContainers`, which is a to-many relationship, is a fault. To fire the fault, get all the objects from the set:

```
Line 1  (lldb) po [[myListObj valueForKey:@"itemContainers"] allObjects]
-       (id) $23 = 0x0d0667b0 <__NSArrayI 0xd0667b0>(
-       <Container: 0x807a180> (entity: Container; id: 0x801bc50 <x-coredata: ↩
            //29B10357-0723-4950-9EB6-E6D7AD6269B9/Container/p1> ; data: {
-           containerId = TestContainer;
5           items = "<relationship fault: 0x805b100 'items'>";
-           state = "(...not nil..)";
-           type = 4;
-       })
-       )
```

Finally, you may be using Transformable attribute types. The state attribute above is an example. If you'd like more information than (. . . not nil. . .), use the KVC getter and you'll see that it's an empty `NSDictionary`:

```
Line 1  (lldb) po [[[[myListObj valueForKey:@"itemContainers"] allObjects]  ↩
            lastObject] valueForKey:@"state"]
-       (id) $29 = 0x0807dd30 {
-       }
-
5       (lldb) po [[[[[myListObj valueForKey:@"itemContainers"] allObjects]  ↩
            lastObject] valueForKey:@"state"] class]
-       (id) $30 = 0x01978e0c __NSCFDictionary
```

2.4 Useful Tools

This section will introduce some interesting tools available to iOS developers to help them code defensively.

Network Link Conditioner

To test your applications in your simulator under different network conditions, you can use the Network Link Conditioner, available as a separate download from Apple.

Get the Hardware IO Tools for Xcode. To do this, go into the Xcode menu, then choose "Open Developer Tool" and finally "More Developer Tools...". You'll be taken to Apple's developer downloads site; you should download the "Hardware IO Tools for Xcode".

The resulting disk image will contain (amongst other things) a preference pane called "Network Link Conditioner". Double-click the prefpane file and authenticate to allow it to be installed. You'll then see the pane in System Preferences, as shown in Figure 2.6.

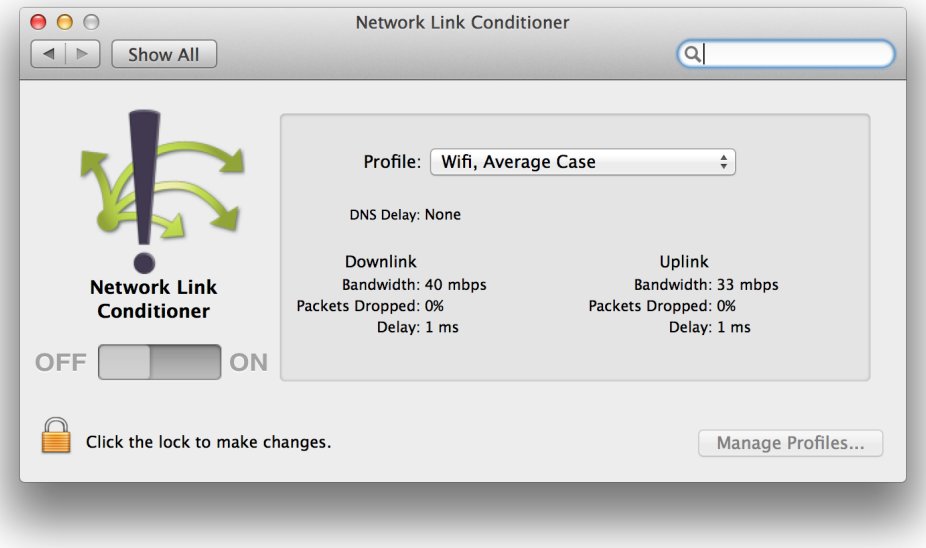

Figure 2.6: Network Link Conditioner preference pane

You can choose from various different types of network conditions using the Profile popup menu, shown in Figure 2.7.

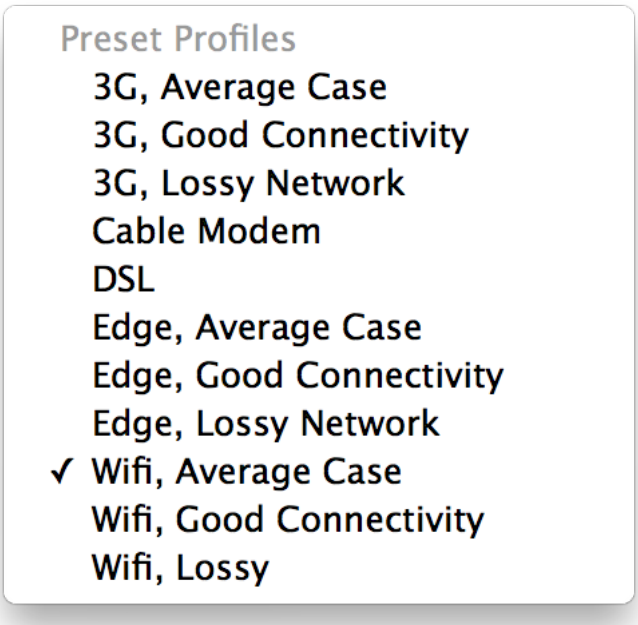

Figure 2.7: Types of network conditions

QuincyKit

QuincyKit is an open source system for automatic crash reporting, composed of client iOS and OS X clients and a PHP + MySQL server backend application. The client frameworks are able to report, upon restart, a complete crash report to the backend application, as shown in Figure 2.8.

Figure 2.8: QuincyKit prompt to send a crash report to the server

To install a working version of QuincyKit, the following steps are required:

Server application:

1. Clone the project from GitHub.
2. Point the local web server to the `server` folder located in the source code distribution.
3. Create a new database in the local MySQL server, named `quincy`.
4. Execute the `database_schema.sql` SQL script to create the required structure in the database.
5. Modify the parameters in the `config.php` file, corresponding to those of your server (around line 70)

```
Line 1  $server = 'localhost';
    -   $loginsql = 'root';
    -   $passsql = 'root';
    -   $base = 'quincy';
```

6. Navigate to `http://localhost:8888/test_setup.php` to make sure that your installation is properly configured.
7. Navigate to `http://localhost:8888/admin/` to access the administration interface.
8. Create an application, specifying the same bundle identifier as your application (as shown in Figure 2.9, in our example the identifier is `com.akosma.QuincyTest`). Specify a name for the application as well.
9. Click on the application name to access the crash report section, shown in Figure 2.10.
10. Define an application version, in this case version 1.0.

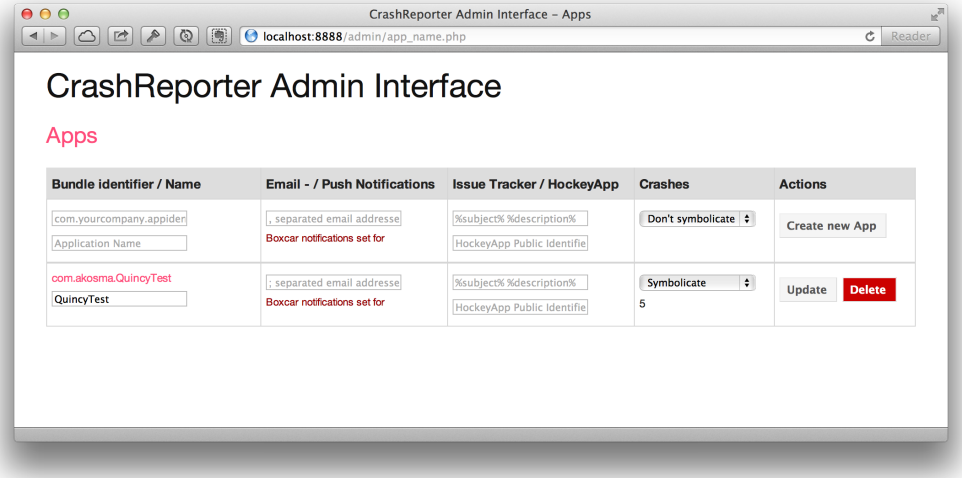

Figure 2.9: QuincyKit administration interface

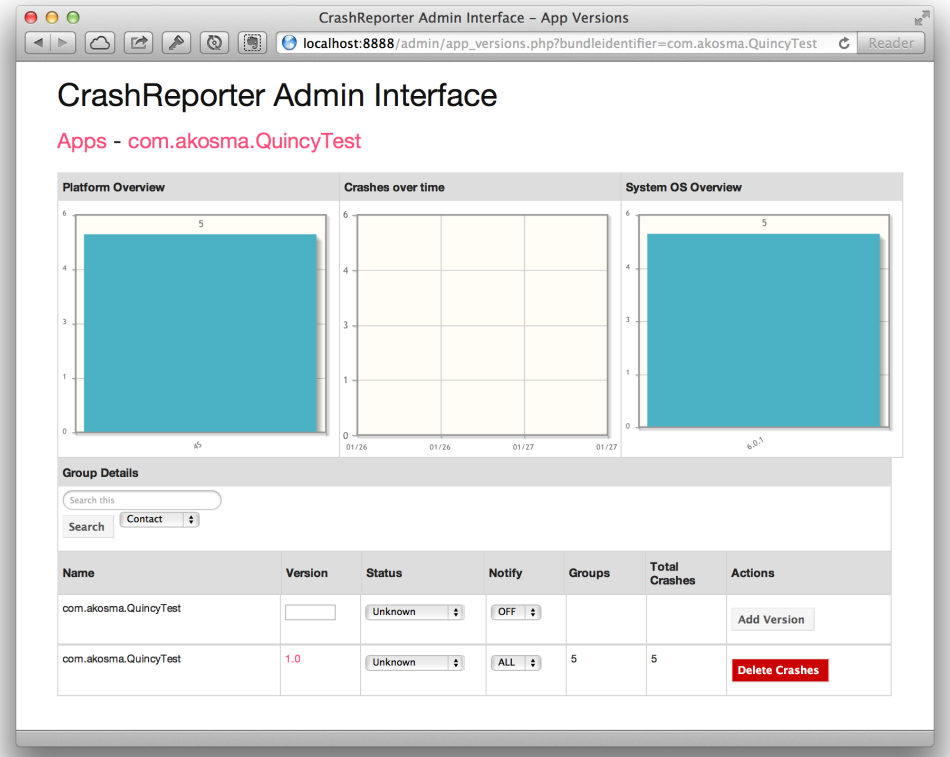

Figure 2.10: Crash reports shown by QuincyKit

Client framework:

1. Copy the following files in your project:
   - BWQuincyManager.h
   - BWQuincyManager.m
   - CrashReporter.framework
   - Quincy.bundle
2. Add the `SystemConfiguration.framework` to your target.
3. If you are using ARC, set the `-fno-objc-arc` flag on the `BWQuincyManager.m` file, in the "Build Phases / Compile Sources" section of the configuration for your target.
4. In the "Build Settings" of your target, add the `-all_load` flag in the "Other Linker Flags" entry.

5. Open your application delegate class, and make it conform to the `BWQuincyManagerDelegate` protocol:

```
Line 1  #import <UIKit/UIKit.h>
    -   #import "BWQuincyManager.h"
    -
    -   @interface AKOAppDelegate : UIResponder <UIApplicationDelegate, ↩
            BWQuincyManagerDelegate>
    5   //...
```

6. In the implementation of your application delegate, include a new line in your `didFinishLaunchingWithOptions:` method:

```
Line 1  @implementation AKOAppDelegate
    -
    -   - (BOOL)application:(UIApplication *)application
    -     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
    5   {
    -       [[BWQuincyManager sharedQuincyManager]
    -                   setSubmissionURL:@"http://servername:8888/ ↩
                            crash_v200.php"];
    -   }
```

Note

The creators of QuincyKit have created a hosted solution called HockeyApp, available for a monthly fee, which can be used without having to host its own server infrastructure.

QuincyKit can also symbolicate crash logs, but this requires a more complex setup, taking care of the following steps:

- In the latest versions of Xcode, the `symbolicatecrash` tool is located at the following (long!) path: `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.pla Developer/Library/PrivateFrameworks/DTDeviceKit.framework/Versions/A/Resources/symbolicatecrash`
- Make sure to add the following variable in your environment before running `symbolicatecrash`: `export DEVELOPER_DIR=`xcode-select --print-path``
- Copy the `.app` and `.app.dSYM` files in a folder, so that QuincyKit can find them and symbolicate the crash logs automatically.

QuincyKit can even be configured to push notifications to developers whenever a certain number of crashes is met, via email or using Growl.

Note

Check Technical Note TN2151 to understand and analyze iOS crash reports.

NSLogger

NSLogger is a very powerful open source utility that allows developers to monitor in real time the log messages of their applications, as testers of beta users interact with the application in a local network. Applications that include the NSLogger code can interact automatically with a desktop OS X application, displaying not only text but also binary data such as images.

Figure 2.11 shows a typical NSLogger session.

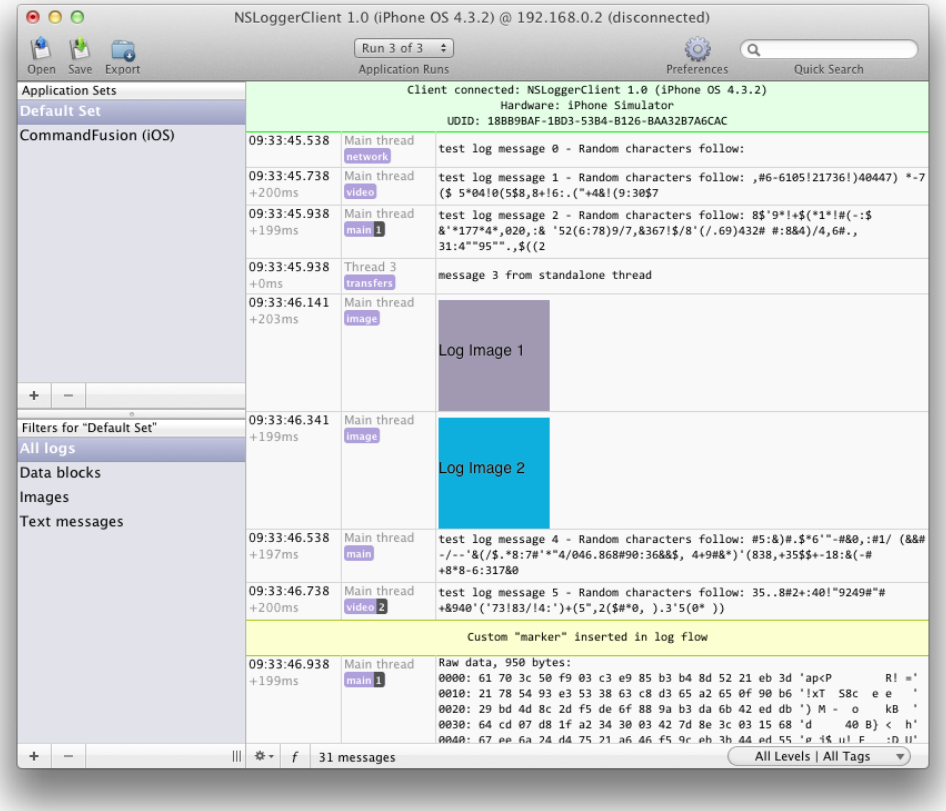

Figure 2.11: NSLogger session

To include NSLogger in an Xcode project, follow these steps:

1. Add the following files to your project:
   - LoggerCommon.h
   - LoggerClient.h
   - LoggerClient.m
2. Add the required system frameworks:
   - CFNetwork.framework
   - SystemConfiguration.framework
3. Use the NSLogger API calls to log stuff:
   - `LogMessage()` to output text;
   - `LogData()` to output raw binary data;
   - `LogImageData()` to output images;
   - `LogMarker()` to output a marker (arbitrary separator in the logger output.

The desktop OS X application is bundled as an Xcode project, that can be built and run off the box. When the application detects a new instance of the application running, it opens a new log window, and each session can be saved and viewed separately. The log output can be filtered and searched very easily.

2.5 Conclusion

Several techniques are available these days for iOS developers to increase the quality of their code; using them will help your teams increase their productivity and the quality of their products.

3 Unit Testing iOS Applications

Unit testing is primary mechanism to ensure that the individual components of an application work properly. It is a very important of testing, albeit not the only one. In this chapter we will see how to use two different mechanisms to test iOS applications.

3.1 OCUnit / SenTest

The OCUnit or SenTest framework, created by the Swiss company Sen:te and included in Xcode since 2004, is the primary mechanism to add unit tests to a project.

Adding Tests

It is very easy to use; whenever you create a new project, Xcode prompts the developer whether to add unit tests to it, as shown in Figure 3.1.

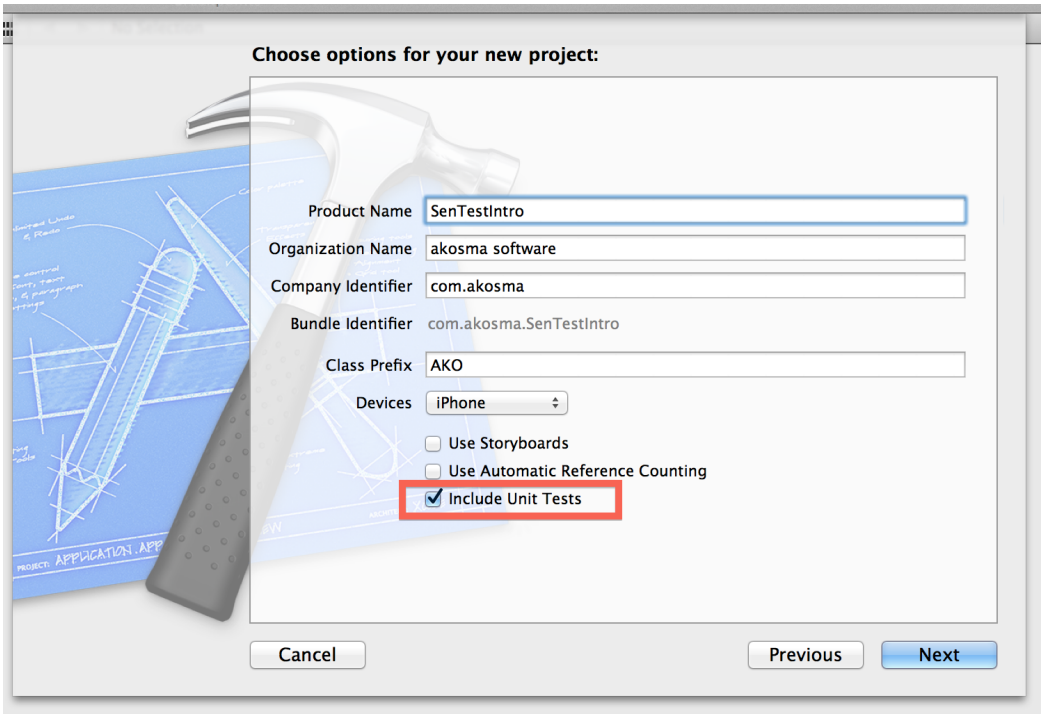

Figure 3.1: Adding tests to an Xcode project

This operation adds a new target to the project, which will include the code to be tested, as well as one or many subclasses of the `SenTestCase` class. In turn, subclasses of `SenTestCase` include one or many methods whose name start with the lowercase `test` word:

```
Line 1  #import <SenTestingKit/SenTestingKit.h>
        -
        -  @interface SenTestIntroTests : SenTestCase
        -
      5 @end
```

The implementation of the test contains the `test...` methods, each including one or more calls to the different `STAssert()` macros available:

```
Line 1  #import "SenTestIntroTests.h"
        -  #import "AKOIntegerCalculator.h"
        -
        -  @interface SenTestIntroTests ()
      5
        -  @property (nonatomic, retain) AKOIntegerCalculator *calc;
        -
```

```
@end


@implementation SenTestIntroTests

- (void)setUp
{
    [super setUp];

    self.calc = [[AKOIntegerCalculator alloc] init];
    NSInteger lastResult = self.calc.lastResult;
    STAssertEquals(lastResult, 0, @"The last result should be 0");
}

- (void)tearDown
{
    self.calc = nil;

    [super tearDown];
}

- (void)testCalcShouldNotBeNil
{
    STAssertNotNil(self.calc, @"The calculator should not be nil");
}

- (void)testCalcCanAdd
{
    NSInteger result = [self.calc add:3 with:5];
    STAssertEquals(result, 3 + 5, @"The result should be the sum of 3 ↩
        and 5");
}

- (void)testCalcCanSubstract
{
    NSInteger result = [self.calc substract:3 by:5];
    STAssertEquals(result, 3 - 5, @"The result should be the 3 minus  ↩
       5");
}

- (void)testCalcCanMultiply
{
    NSInteger result = [self.calc multiply:3 by:5];
```

```
    STAssertEquals(result, 3 * 5, @"The result should be the 3 times 5");
}

- (void)testCalcCanDivide
{
    NSInteger result = [self.calc divide:8 by:5];
    STAssertEquals(result, 8 / 5, @"The result should be the 8 divided by 5");
}

- (void)testCalcRaisesExceptionIfDivideByZero
{
    STAssertThrows([self.calc divide:4 by:0], @"Attempt to divide by zero");
}

@end
```

Here you have the complete list of assertion macros you can use in OCUnit:

- STAssertNil(a1, description, ...)
- STAssertNotNil(a1, description, ...)
- STAssertTrue(expression, description, ...)
- STAssertFalse(expression, description, ...)
- STAssertEqualObjects(a1, a2, description, ...)
- STAssertEquals(a1, a2, description, ...)
- STAssertEqualsWithAccuracy(left, right, accuracy, description, ...)
- STAssertThrows(expression, description, ...)tNoThrowSpecificNamed(expr, exception, aName, desc, ...)
- STFail(description, ...)
- STAssertTrueNoThrow(expression, description, ...)
- STAssertFalseNoThrow(expression, description, ...)

Running Tests

To execute the unit tests, just select "Product / Test" or hit the "Command-U" keystroke in Xcode. The output of the execution of the tests is shown below:

```
Line 1  Test Suite '-[SenTestIntroTests' started at 2013-01-28 15:08:36 +0000
-       Test Case '-[SenTestIntroTests testCalcCanAdd]' started.
-       Test Case '-[SenTestIntroTests testCalcCanAdd]' passed (0.000 seconds).
-       Test Case '-[SenTestIntroTests testCalcCanDivide]' started.
5       Test Case '-[SenTestIntroTests testCalcCanDivide]' passed (0.000 seconds).
-       Test Case '-[SenTestIntroTests testCalcCanMultiply]' started.
-       Test Case '-[SenTestIntroTests testCalcCanMultiply]' passed (0.000 seconds).
-       Test Case '-[SenTestIntroTests testCalcCanSubstract]' started.
-       Test Case '-[SenTestIntroTests testCalcCanSubstract]' passed (0.000 seconds).
10      Test Case '-[SenTestIntroTests testCalcRaisesExceptionIfDivideByZero]' started.
-       Test Case '-[SenTestIntroTests testCalcRaisesExceptionIfDivideByZero]' passed (0.000 seconds).
-       Test Case '-[SenTestIntroTests testCalcShouldNotBeNil]' started.
-       Test Case '-[SenTestIntroTests testCalcShouldNotBeNil]' passed (0.000 seconds).
-       Test Suite '-[SenTestIntroTests' finished at 2013-01-28 15:08:36 +0000.
15      Executed 6 tests, with 0 failures (0 unexpected) in 0.000 (0.001) seconds
```

Functional Testing

Using OCUnit you can also test controllers, thus providing a higher level of interaction with your code, which makes it very similar to using the functional testing techniques described in Chapter 4.

To do this, we add a new file of type "Objective-C Test Case" to our project, and we write the code required:

```
Line 1  #import "AKOViewControllerTest.h"
-       #import "AKOViewController.h"
-
-       @interface AKOViewControllerTest ()
5
-       @property (nonatomic, retain) AKOViewController *vc;
-
-       @end
-
10
```

```
@implementation AKOViewControllerTest

- (void)setUp
{
    [super setUp];

    self.vc = [[AKOViewController alloc] init];

    // This trick force loads the NIB
    [self.vc view];
}

- (void)tearDown
{
    self.vc = nil;

    [super tearDown];
}

- (void)testHasBlankDisplay
{
    STAssertTrue([self.vc.displayLabel.text isEqualToString:@"0"], @" ↩
        Upon start, the display should be empty");
}

- (void)testAdd123To456
{
    [self.vc enter:self.vc.button1];
    [self.vc enter:self.vc.button2];
    [self.vc enter:self.vc.button3];
    STAssertTrue([self.vc.displayLabel.text isEqualToString:@"123"],  ↩
        @"The display should say 123");

    [self.vc performAdd:nil];
    STAssertTrue([self.vc.displayLabel.text isEqualToString:@"0"], @" ↩
        The display should say 0");

    [self.vc enter:self.vc.button4];
    [self.vc enter:self.vc.button5];
    [self.vc enter:self.vc.button6];
    STAssertTrue([self.vc.displayLabel.text isEqualToString:@"456"],  ↩
        @"The display should say 456");

```

```
50      [self.vc performEqual:nil];
-       STAssertTrue([self.vc.displayLabel.text isEqualToString:@"579"],  ↩
            @"The display should say 579");
-   }
-
-   @end
```

In the code above we simulate a manual interaction on the view controller, executing the `IBAction` methods in the required order. At every time, we check the values of the display of the calculator, making sure that it displays the correct values at every step.

Note

The test shown above will be repeated throughout this book; to compare the different frameworks, we are going to create a simple test that adds 123 to 456, yielding a value of 579, as expected.

3.2 Kiwi

Kiwi is a very popular new library that enables iOS developers to use BDD ("Behavior Driven Development") techniques in their workflow.

Kiwi is built on top of OCUnit (described in the previous section) and uses blocks extensively, making it very similar to similar libraries such as RSpec for Ruby or Jasmine for JavaScript.

Note

At the time of this writing, the latest documented and stable version of Kiwi is version 1.1. Version 2.0 is already released, but it lacks documentation and will not be used in this section.

Adding Kiwi to a project

We are going to add Kiwi to the calculator project; for that, follow these steps:

1. Download the source code of Kiwi from Github.
2. Copy the source files into your project, in a folder / group named "Kiwi"
3. Create a unit testing target in your project, if it is not available.

Adding Specs to a Project

To add a KIWI specification to the project, add a C file (no header file required) and add the code below:

```
Line 1  #import "Kiwi.h"
-       #import "AKOIntegerCalculator.h"
-
-       SPEC_BEGIN(CalculatorSpec)
5
-       describe(@"The calculator", ^{
-
-           context(@"when created", ^{
-
10             __block AKOIntegerCalculator *calc = nil;
-
-              beforeEach(^{
-                  calc = [[AKOIntegerCalculator alloc] init];
-              });
15
-              it(@"is not nil", ^{
-                  [calc shouldNotBeNil];
-              });
-
20            afterEach(^{
-                  [calc release];
-              });
-
-          });
25
-          context(@"when calculating", ^{
-
-              __block AKOIntegerCalculator *calc = nil;
-
30            beforeAll(^{
-                  calc = [[AKOIntegerCalculator alloc] init];
-              });
-
-              beforeEach(^{
35                NSInteger lastResult = calc.lastResult;
-                 [[theValue(lastResult) should] equal:theValue(0)];
-             });
-
-             it(@"can add", ^{
```

```
40              NSInteger result = [calc add:3 with:5];
-               [[theValue(result) should] equal:theValue(3 + 5)];
-           });
-
-           it(@"can substract", ^{
45              NSInteger result = [calc substract:3 by:5];
-               [[theValue(result) should] equal:theValue(3 - 5)];
-           });
-
-           it(@"can multiply", ^{
50              NSInteger result = [calc multiply:3 by:5];
-               [[theValue(result) should] equal:theValue(3 * 5)];
-           });
-
-           it(@"can divide", ^{
55              NSInteger result = [calc divide:8 by:5];
-               [[theValue(result) should] equal:theValue(8 / 5)];
-           });
-
-           it(@"raises an exception if dividing by zero", ^{
60              [[theBlock(^{
-                   NSInteger result = [calc divide:4 by:0];
-                   NSLog(@"result: %d", result);
-               }) should] raiseWithName:@" ↩
                    AKOIntegerCalculatorDivideByZero"];
-           });
65
-           afterEach(^{
-               [calc reset];
-           });
-
70         afterAll(^{
-              [calc release];
-          });
-
-      });
75
- });
-
- SPEC_END
```

Note the `SPEC_BEGIN` and `SPEC_END` macros. The preprocessor uses them to build the interface and implementation of a normal `SenTestCase` subclass.

SPEC_BEGIN's sole argument is the actual subclass name. It needs to be a valid class identifier and a unique symbol in the application.

The strings passed to describe and it macros become part of the test output and error reporting when the tests fail. The should method returns an object that responds to matcher methods like the equal: above. (The should method is added to NSObject so anything can start triggering an assertion.)

The beforeEach method is analogous to the setUp in SenTestCase subclasses. As you can imagine, afterEach is like tearDown.

You can nest describe blocks within other describe blocks. Using block scope you can share test state with the nested describe's to help cut down on ceremonial code noise.

If you need to modify the values of variables defined outside of blocks you need to use the __block modifier, which will make the variables read/write instead of just readonly.

Kiwi is a large framework bolted on top of SenTestingKit. It depends a lot on the magic behind the Objective C runtime. It adds should and shouldNot onto NSObject. It does a lot of fancy selector forwarding to get the matchers to work.

As shown in the code above, the tests are much more descriptive than those created using OCUnit; they actually deliberately explain the purpose and logic of each scenario of use of the application, in a language that is much closer to what humans can understand.

Testing User Interfaces with Kiwi

We can also test view controllers with Kiwi, in a very similar way to OCUnit.

```
Line 1  #import "Kiwi.h"
    -   #import "AKOViewController.h"
    -
    -   SPEC_BEGIN(AKOViewControllerSpec)
    5
    -   describe(@"The calculator view controller", ^{
    -       context(@"when it starts", ^{
    -
    -           __block AKOViewController *vc = nil;
   10
    -           beforeAll(^{
    -               vc = [[AKOViewController alloc] init];
    -
    -               // This trick force loads the NIB
```

```
            [vc view];
        });

        it(@"has a blank display", ^{
            [[vc.displayLabel.text should] equal:@"0"];
        });

        it(@"displays 579 if the user adds 123 and 456", ^{
            [vc enter:vc.button1];
            [vc enter:vc.button2];
            [vc enter:vc.button3];
            [[vc.displayLabel.text should] equal:@"123"];

            [vc performAdd:nil];
            [[vc.displayLabel.text should] equal:@"0"];

            [vc enter:vc.button4];
            [vc enter:vc.button5];
            [vc enter:vc.button6];
            [[vc.displayLabel.text should] equal:@"456"];

            [vc performEqual:nil];
            [[vc.displayLabel.text should] equal:@"579"];
        });

        afterAll(^{
            [vc release];
        });

    });
});

SPEC_END
```

The functional tests above, as usual, test whether a simple calculation is performed without problem by the software.

3.3 BDD vs TDD

In this chapter we have studied how to use TDD and BDD to increase the quality of our applications, which raises the question: what is the main difference between those two techniques? Well, it turns out that there is almost none.

Dan North has said that

> TDD – as originally described – is also about the behaviour of entire systems. Kent [Beck] specifically describes it as operating on multiple levels of abstraction, not just "down in the code". BDD is equally important in this space, because describing the behaviour of systems is fractal: you can describe different granularities of behaviour from the entire application right down to individual small components, classes or functions.

When Dan was working as a coach teaching TDD, he found that it was easier to get people to understand the principles of TDD if he stopped using the word 'test':

> My experiences as a coach told me people were missing the point, with all this talk of unit tests, acceptance tests, functional tests, integration tests... Kent Beck's style of TDD is a very smart way to develop software, so I tried removing the word "test" when I was coaching it, replacing it with things like behaviour, examples, scenarios etc. The result was very encouraging: People seemed to "get" TDD much quicker when I avoided referring to testing.

BDD re-explains TDD in a way that highlights the habits that successful TDD practitioners having been using since the end of the 90's:

1. Working outside-in, starting from a business or organisational goal
2. Using examples to clarify requirements
3. Developing and using a ubiquitous language

Working outside-in seems obvious to habitual TDD practitioners, but many teams seem to limit themselves to doing this at the level of small units of code. Business-level black-box testing is still done manually, or automated as a check after the code has already been implemented.

This misses out of the major benefit of working outside-in, which is having the requirement challenged: if you need to explain to a computer how to check the requirement, you'll need to be damn sure understand it yourself. If you don't (and you often don't) it's much cheaper to find that out before you write the code.

What BDD does is formalise this by encouraging you to use scenarios to describe behaviour. These examples provide the perfect bridge between the business-facing and technology-facing sides of a team: they're just formal enough that you can get a computer to check them, but anyone on the team can read them and make sure they're describing behaviour that they actually want.

BDD's emphasis on collaboration, and the use of business-readable, executable specifications, means that this shared language develops much more quickly. When everyone is involved in writing documentation that describes what the system should do, they all get a chance to learn the language of the domain together.

So BDD really isn't all that different to TDD. What BDD adds is a clear emphasis on what it takes to make TDD succeed.

3.4 Conclusion

Both TDD and BDD all contribute to provide live documentation and feedback to team members about the overall quality of the code base. Choosing one or the other, or other alternatives such as GHUnit, Google Toolbox for Mac, Specta, Cedar or other tools such as XcodeCoverage, will all have the intended effect of helping you build reliable, solid code.

4 Functional Testing of iOS Applications

This chapter will provide an overview of techniques for UI testing automation of iOS applications using Frank, Calabash-iOS, KIF and finally Apple Instruments.

4.1 Frank

This section will cover the essential steps to get started with Frank, a very useful tool built on top of Cucumber.

Note
At the time of this writing, the latest version of Frank is version 1.1.6.

Getting Started with Frank

First things first. Make sure that you have Cucumber installed in your machine, using RubyGems:

```
Line 1  gem install cucumber
```

Note
At the time of this writing, the latest version of Cucumber is 1.2.1, and the latest version of Calabash-Cucumber is 0.9.129.

Install Frank on top of Cucumber now:

```
Line 1  gem install frank-cucumber
```

You will need to turn on the accessibility features on the machine hosting your iOS simulator. Frank leverages accessibility to automate some actions with the simulator (such as rotating the device).

On the machine which will be hosting the iOS Simulator go to System Preferences → Universal Access and Check “Enable access for assistive devices” as shown in Figure 4.1.

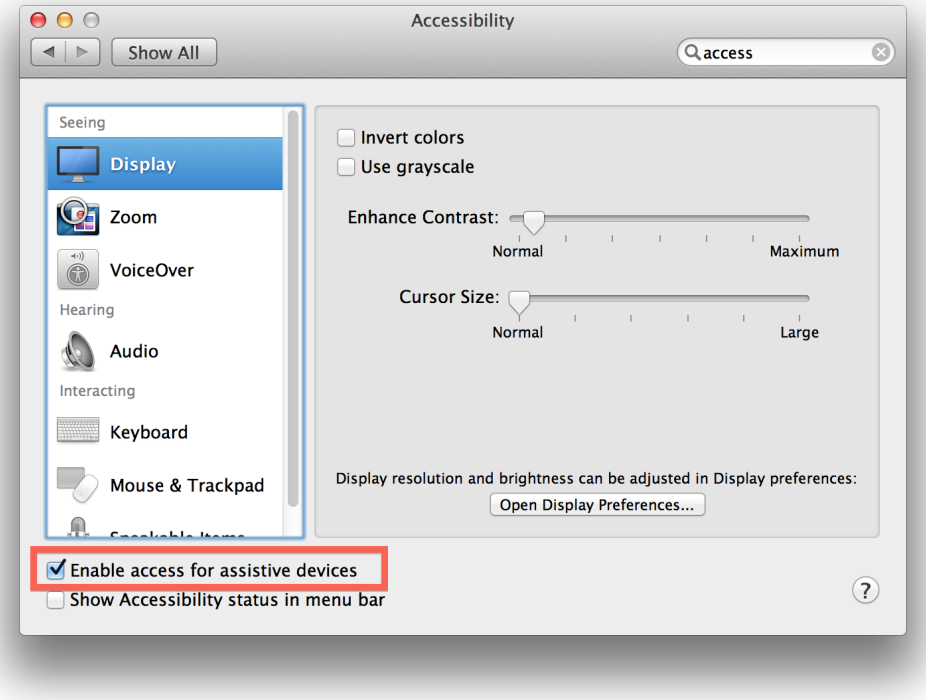

Figure 4.1: Enabling accessibility in OS X

Then you need to "Frankify" your iOS application, which is done by running the following commands in the root folder of your project:

```
Line 1  frank setup
```

If your project contains several targets, the command-line tool will ask you to specify which target should be frankified.

The next command actually builds the frankified version of your application:

```
Line 1  frank build
```

Finally, to run the application, just use the following command, which effectively launches the iOS simulator with your application inside:

```
Line 1  frank launch
```

The last command opens a small embedded web application called "Symbiote". It allows you to inspect the current state of your app as it's running. It also lets you experiment with view selectors. View selectors are how you specify which views in your app you want to interact with or inspect the value of. If you're familiar with CSS selectors or XPath expressions the concept is the same.

```
Line 1  frank inspect
```

You can see Symbiote running in Figure 4.2.

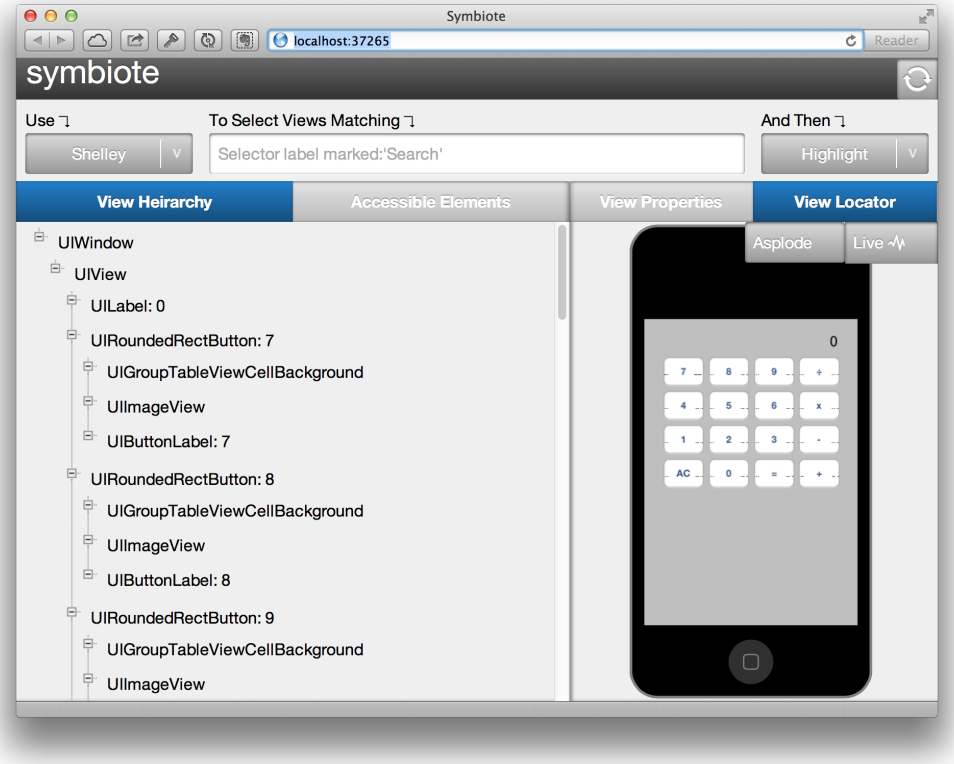

Figure 4.2: The Symbiote application launched by Frank

Adding Custom Tests

Our goal is to use Frank and cucumber to run automated tests against our app. We can run the initial cucumber test that was provided as part of `frank setup`. To do that simple run `cucumber` in a terminal from the Frank subdirectory. When you do that you should see the Frankified app launch in the simulator, and then perform some rotations. You'll also see some output in the terminal from cucumber describing the test steps it has performed and eventually declaring the test scenario passed.

This is the output shown in the console window:

```
Line 1  Feature:
-         As an iOS developer
-         I want to have a sample feature file
-         So I can see what my next step is in the wonderful world of Frank/ ↩
              Cucumber testing
5
-       Scenario:                                          # features/ ↩
            my_first.feature:6
-           Rotating the simulator for demonstration purposes
-         Given I launch the app                           # features/ ↩
              step_definitions/launch_steps.rb:5
-         Given the device is in landscape orientation     # frank- ↩
              cucumber-1.1.6/lib/frank-cucumber/core_frank_steps.rb:151
10        Given the device is in portrait orientation      # frank- ↩
              cucumber-1.1.6/lib/frank-cucumber/core_frank_steps.rb:151
-         Given the device is in landscape orientation     # frank- ↩
              cucumber-1.1.6/lib/frank-cucumber/core_frank_steps.rb:151
-         Given the device is in portrait orientation      # frank- ↩
              cucumber-1.1.6/lib/frank-cucumber/core_frank_steps.rb:151
-
-       1 scenario (1 passed)
15     5 steps (5 passed)
-      0m4.794s
```

Now that we know how to run cucumber tests we should write our own. We'll write these tests in a new feature file called `Frank/features/typing.feature`. Create that file with the following content:

```
Line 1  Feature: Typing numbers
-
-           Scenario: Typing 123 in the keyboard
-               Given I launch the app
```

```
5           Then I should see zero in the display
-
-           When I touch the button marked "1"
-           Then I should see "1" in the display
-
10          When I touch the button marked "2"
-           Then I should see "12" in the display
-
-           When I touch the button marked "3"
-           Then I should see "123" in the display
15
-           When I touch the button marked "+"
-           Then I should see zero in the display
-
-           When I touch the button marked "4"
20          Then I should see "4" in the display
-
-           When I touch the button marked "5"
-           Then I should see "45" in the display
-
25          When I touch the button marked "6"
-           Then I should see "456" in the display
-
-           When I touch the button marked "="
-           Then I should see "579" in the display
```

This expresses a test scenario. Now let's ask cucumber to test just this feature by running `cucumber features/typing.feature`. You should see the app launch, but then cucumber will complain because it doesn't know how to execute any of the steps we've described after launching the app. That's fair enough; we haven't defined them anywhere yet! Let's do that now.

Create a step definition file called `features/step_definitions/typing_steps.rb`. When cucumber encountered the undefined steps just now it outputted a bunch of boilerplate code for defining those steps. We'll cut and paste that code into our new step definition file. You should end up with this:

```
Line 1  Then /^I should see zero in the display$/ do
-           check_element_exists("view:'UILabel' marked:'0'")
-       end
-
5       Then /^I should see "(.*?)" in the display$/ do |arg1|
-           check_element_exists("view:'UILabel' marked:'#{arg1}'")
```

```
end
```

Finally, running `cucumber features/typing.feature` should yield the following output:

Feature: Typing numbers

```
  Scenario: Typing 123 in the keyboard
    Given I launch the app
    Then I should see zero in the display
    When I touch the button marked "1"
    Then I should see "1" in the display
    When I touch the button marked "2"
    Then I should see "12" in the display
    When I touch the button marked "3"
    Then I should see "123" in the display
    When I touch the button marked "+"
    Then I should see zero in the display
    When I touch the button marked "4"
    Then I should see "4" in the display
    When I touch the button marked "5"
    Then I should see "45" in the display
    When I touch the button marked "6"
    Then I should see "456" in the display
    When I touch the button marked "="
    Then I should see "579" in the display

1 scenario (1 passed)
18 steps (18 passed)
0m15.130s
```

4.2 Calabash-iOS

Calabash-iOS is also based on Cucumber like Frank, but its license is more permissive and commercial-friendly (Frank is released through the GPL while Calabash uses the Eclipse license.)

Note

At the time of this writing, the latest version of Calabash-Android is version 0.3.8.

Getting Started

To use Calabash, install the gem:

```
Line 1  gem install calabash-cucumber
```

Similarly as Frank, you need to prepare your project to use Calabash:

```
Line 1  calabash-ios setup
```

Warning
Make sure that Xcode is not running before using the `calabash-ios setup` command.

Then, generate skeleton features for your tests:

```
Line 1  calabash-ios gen
```

Next, you must open your Xcode project, and build the application using the scheme that ends with `-cal` in its name. Finally, run the tests:

```
Line 1  cucumber
```

Adding Tests

As usual, we are going to create a very simple test that reproduces a common interaction with the calculator: the addition of 123 and 456.

This is the test required:

```
Line 1  Feature: Typing numbers
-       This is a calculator
-       so I want to add 123 to 456
-       and then see the results.
5
-   Scenario: Calculate
-       Given I am on the Welcome Screen
-       Then I touch "1"
-       Then I touch "2"
10      Then I touch "3"
-       Then I touch "+"
-       Then I touch "4"
-       Then I touch "5"
-       Then I touch "6"
```

```
    Then I touch "="
    Then I wait to see "579"
    And take picture
```

The output of this command is shown below:

```
Feature: Typing numbers
    This is a calculator
    so I want to add 123 to 456
    and then see the results.

  Scenario: Calculate                # features/typing.feature:6
Waiting at most 30 seconds for simulator (CONNECT_TIMEOUT)
Retrying at most 2 times (MAX_CONNECT_RETRY)
(1.) Start Simulator 6.0, iphone, for /Users/adrian/Library/Developer ↩
    /Xcode/DerivedData/CalabashIntro-aacwdpfznewgjscipwldvplwdvfx/ ↩
    Build/Products/Debug-iphonesimulator/CalabashIntro-cal.app
Ping http://localhost:37265/...
Fetch version http://localhost:37265/version...
{"outcome"=>"SUCCESS", "app_name"=>"CalabashIntro-cal", " ↩
    simulator_device"=>"iPhone", "iOS_version"=>"6.0", "app_version" ↩
    =>"1.0", "system"=>"x86_64", "app_id"=>"com.akosma.CalabashIntro- ↩
    cal", "version"=>"0.9.126", "simulator"=>"iPhone Simulator 358.4, ↩
     iPhone OS 6.0 (iPhone (Retina 3.5-inch)/10A403)"}
    Given I am on the Welcome Screen
    Then I touch "1"
    Then I touch "2"
    Then I touch "3"
    Then I touch "+"
    Then I touch "4"
    Then I touch "5"
    Then I touch "6"
    Then I touch "="
    Then I wait to see "579"
    And take picture

1 scenario (1 passed)
11 steps (11 passed)
0m7.993s
```

Note

A very similar tool exists for Android applications as well, called Calabash-Android. It will be described in Chapter 7.

4.3 KIF

KIF, which stands for Keep It Functional, is an iOS integration test framework. It allows for easy automation of iOS apps by leveraging the accessibility attributes that the OS makes available for those with visual disabilities.

KIF uses undocumented Apple APIs. This is true of most iOS testing frameworks, and is safe for testing purposes, but it's important that KIF does not make it into production code, as it will get your app submission denied by Apple. Follow the instructions below to ensure that KIF is configured correctly for your project.

Features

- Minimizes Indirection: All of the tests for KIF are written in Objective C. This allows for maximum integration with your code while minimizing the number of layers you have to build.
- Easy Configuration: KIF integrates directly into your iOS app, so there's no need to run an additional web server or install any additional packages.
- Test Like a User: KIF attempts to imitate actual user input. Automation is done using tap events wherever possible.

Installation

To install KIF, follow these steps:

1. Use git submodules to add the KIF library to your project:

```
Line 1  mkdir Frameworks git submodule add https://github.com/ ↩
            square/KIF.git
    -   Frameworks/KIF
```

2. Create a workspace that holds both your application and the KIF project.
3. Duplicate the target of your application so that there is a separate binary to test (this is required since KIF uses private APIs that might get your application rejected by Apple!)
4. Now that you have a target for your tests, add the tests to that target. With the project settings still selected in the Project Navigator, and the new integration tests target selected in the project settings, select the "Build Phases" tab. Under the "Link Binary With Libraries" section, hit the "+"

button. In the sheet that appears, select "libKIF.a" and click "Add.", as shown in Figure 4.3.

5. Next, make sure that we can access the KIF header files. To do this, add the KIF directory to the "Header Search Paths" build setting. Start by selecting the "Build Settings" tab of the project settings, and from there, use the filter control to find the "Header Search Paths" setting. Double click the value, and add the search path "$(SRCROOT)"/Frameworks/KIF/ to the list (do not forget the quotes around "$(SRCROOT)"!). Mark the entry as recursive. If it's not there already, you should add the $(inherited) entry as the first entry in this list.

6. KIF takes advantage of Objective C's ability to add categories on an object, but this isn't enabled for static libraries by default. To enable this, add the -ObjC and -all_load flags to the "Other Linker Flags" build setting.

7. Finally, add a preprocessor flag to the testing target so that you can conditionally include code. This will help make sure that none of the testing code makes it into the production app. Call the flag RUN_KIF_TESTS=1 and add it under the "Preprocessor Macros." Again, make sure the $(inherited) entry is first in the list.

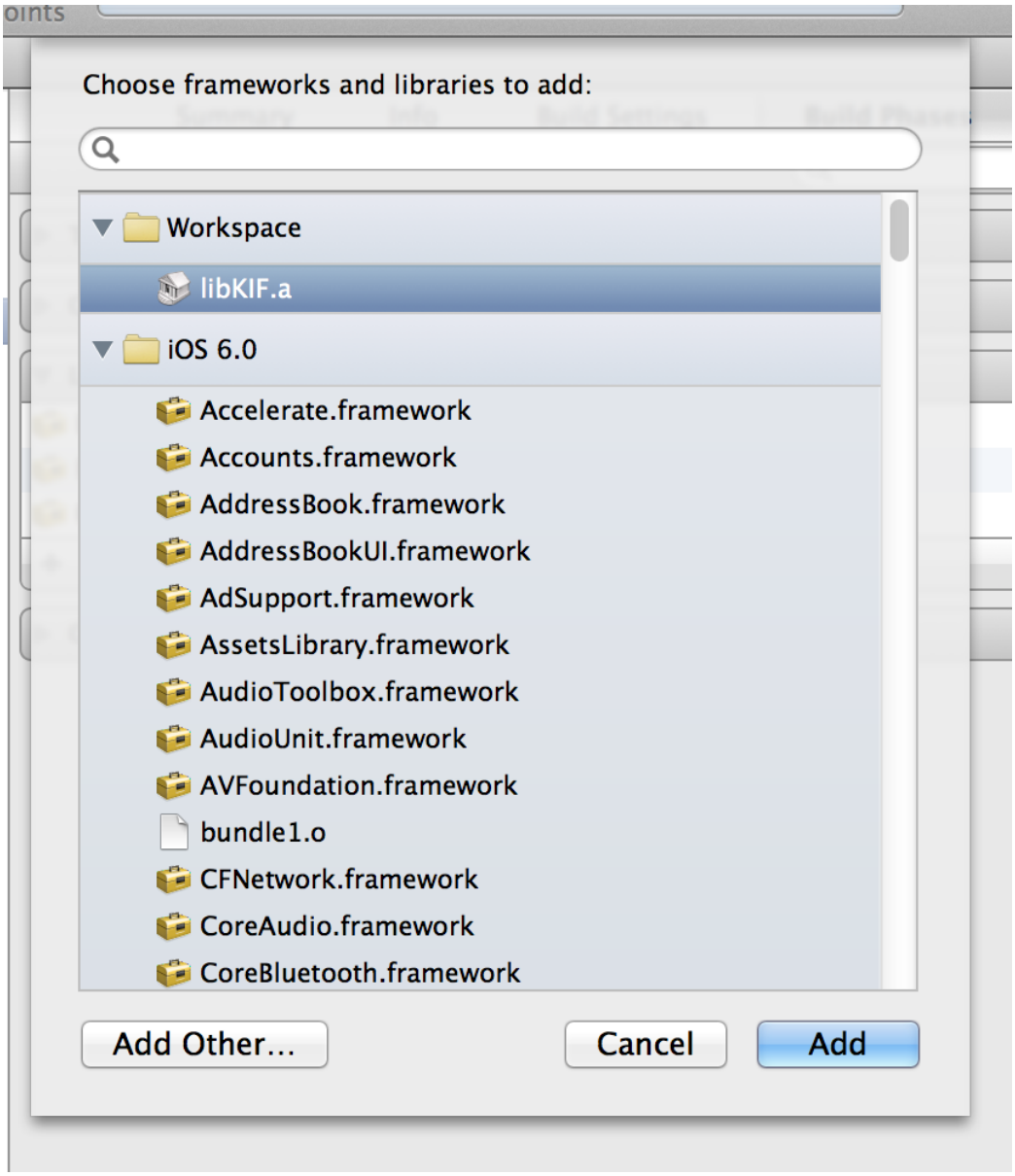

Figure 4.3: Selecting the KIF library

Writing Tests

With your project configured to use KIF, it's time to start writing tests. There are three main classes used in KIF testing: the test runner (KIFTestController), a testable scenario (KIFTestScenario), and a test step (KIFTestStep). The test runner is compoto this view," and "wait for this view." These steps are included as factory methods on KIFTestStep in the base KIF implementation.

KIF relies on the built-in accessibility of iOS to perform its test steps. As such, it's important that your app is fully accessible. This is also a great way to ensure that your app is usable by the sight impaired. Making your application accessible is usually as easy as giving your views reasonable labels.

Although not required, it's recommended that you create a subclass of KIFTestController that is specific to your application. This subclass will override the -initializeScenarios method, which will contain a list of invocations for the scenarios that your test suite will run. We'll call our subclass AKOTestController, and will add an initial test scenario, which we will define later.

The code for the tests follows below:

AKOTestController.h

```
Line 1  #import "KIFTestController.h"
    -
    -   @interface AKOTestController : KIFTestController
    -
    5   @end
```

AKOTestController.m

```
Line 1  #import "AKOTestController.h"
    -   #import "KIFTestScenario+AKOTestScenario.h"
    -
    -   @implementation AKOTestController
    5
    -   - (void)initializeScenarios;
    -   {
    -       [self addScenario:[KIFTestScenario scenarioToCalculate]];
    -   }
   10
    -   @end
```

KIFTestScenario+AKOTestScenario.h

```
Line 1  #import "KIFTestScenario.h"
    -
    -   @interface KIFTestScenario (AKOTestScenario)
    -
    5   + (id)scenarioToCalculate;
    -
    -   @end
```

KIFTestScenario+AKOTestScenario.m

```
Line 1  #import "KIFTestScenario+AKOTestScenario.h"
    -   #import "KIFTestStep+AKOTestStep.h"
    -
    -   @implementation KIFTestScenario (AKOTestScenario)
    5
    -   + (id)scenarioToCalculate
    -   {
    -       KIFTestScenario *scenario = [KIFTestScenario ↩
                scenarioWithDescription:@"Test that a user can make a basic ↩
                calculation"];
    -       [scenario addStepsFromArray:[KIFTestStep stepsToTap123]];
   10       [scenario addStep:[KIFTestStep stepToTapPlus]];
    -       [scenario addStepsFromArray:[KIFTestStep stepsToTap456]];
    -       [scenario addStep:[KIFTestStep stepToTapEqual]];
    -       [scenario addStep:[KIFTestStep stepToCheckResult]];
    -       return scenario;
   15   }
    -
    -   @end
```

KIFTestStep+AKOTestStep.h

```
Line 1  #import "KIFTestStep.h"
    -
    -   @interface KIFTestStep (AKOTestStep)
    -
    5   + (NSArray *)stepsToTap123;
    -   + (id)stepToTapPlus;
    -   + (NSArray *)stepsToTap456;
    -   + (id)stepToTapEqual;
    -   + (id)stepToCheckResult;
   10
    -   @end
```

KIFTestStep+AKOTestStep.m

```
Line 1  #import "KIFTestStep+AKOTestStep.h"
    -   #import "UIApplication-KIFAdditions.h"
    -   #import "UIAccessibilityElement-KIFAdditions.h"
    -
    5   @implementation KIFTestStep (AKOTestStep)
    -
    -   + (NSArray *)stepsToTap123
```

```
{
    NSMutableArray *steps = [NSMutableArray array];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"1"]];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"2"]];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"3"]];
    return steps;
}

+ (id)stepToTapPlus
{
    return [KIFTestStep stepToTapViewWithAccessibilityLabel:@"+"];
}

+ (NSArray *)stepsToTap456
{
    NSMutableArray *steps = [NSMutableArray array];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"4"]];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"5"]];
    [steps addObject:[KIFTestStep stepToTapViewWithAccessibilityLabel:@"6"]];
    return steps;
}

+ (id)stepToTapEqual
{
    return [KIFTestStep stepToTapViewWithAccessibilityLabel:@"="];
}

+ (id)stepToCheckResult
{
    NSString *expectedLabel = @"579";
    NSString *description = [NSString stringWithFormat:@"Verify calculation result"];
    return [self stepWithDescription:description executionBlock:^(KIFTestStep *step, NSError **error) {
        UIAccessibilityElement *element = [[UIApplication sharedApplication] accessibilityElementWithLabel:@"displayLabel"];
```

```
        UILabel *label = (UILabel *)[UIAccessibilityElement
            viewContainingAccessibilityElement:element];
        if ([expectedLabel isEqualToString:label.text])
        {
            return KIFTestStepResultSuccess;
        }

        KIFTestCondition(NO, error, @"Failed to compare the label
            text: expected '%@', actual '%@'", expectedLabel, label.
            text);
    }];
}

@end
```

KIF tests can be automatized to be used in Jenkins, using the WaxSim tool.

4.4 UI Automation with Instruments

The Instruments application, bundled with Xcode, can be used to automate UI functional testing tasks. It uses JavaScript as the language to create the tests.

Note

At the time of this writing, the latest version of Xcode is 4.6.

Creating UI Tests with Instruments

To create UI tests with Instruments, follow these steps:

1. Select the "Product / Profile" menu entry in Instruments and select the "Automation" instrument, as shown in Figure 4.4. This will launch the Instruments application.
2. Click on the "Add" button in the left pane of the Instruments screen, and select "Create" in the small pop-up menu.
3. Double click on the script name to "Typing".
4. Add the following JavaScript code in the editor window:

```
var target = UIATarget.localTarget();
var mainWindow = target.frontMostApp().mainWindow()

mainWindow.buttons()["AC"].tap();
```

```
5
-  mainWindow.buttons().firstWithName("1").tap();
-  mainWindow.buttons().firstWithName("2").tap();
-  mainWindow.buttons().firstWithName("3").tap();
-  mainWindow.buttons()["+"].tap();
10 mainWindow.buttons().firstWithName("4").tap();
-  mainWindow.buttons().firstWithName("5").tap();
-  mainWindow.buttons().firstWithName("6").tap();
-  mainWindow.buttons()["="].tap();
-
15 var display = mainWindow.staticTexts()[0];
-
-  if (display.value() === "579") {
-      UIALogger.logPass("Typing");
-  }
20 else {
-      UIALogger.logFail("Typing");
-  }
```

Then click on the "Play" button that appears at the bottom of the editor window, and the UI of the application in the simulator will reproduce the steps of the test, and will mark the test as passed, as shown in Figure 4.5.

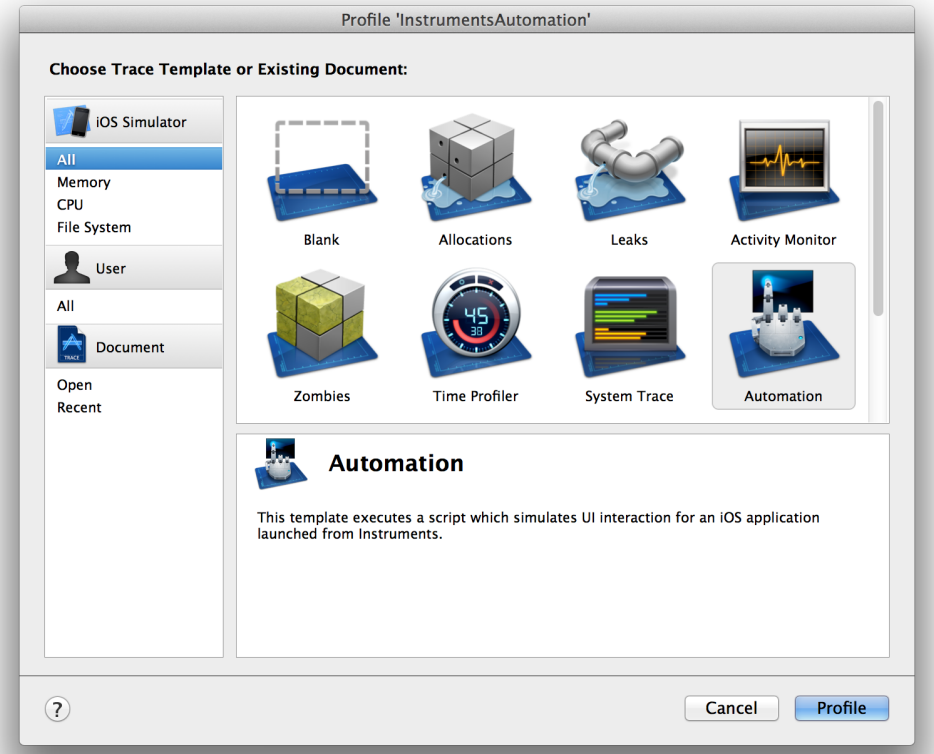

Figure 4.4: Selecting the automation instrument

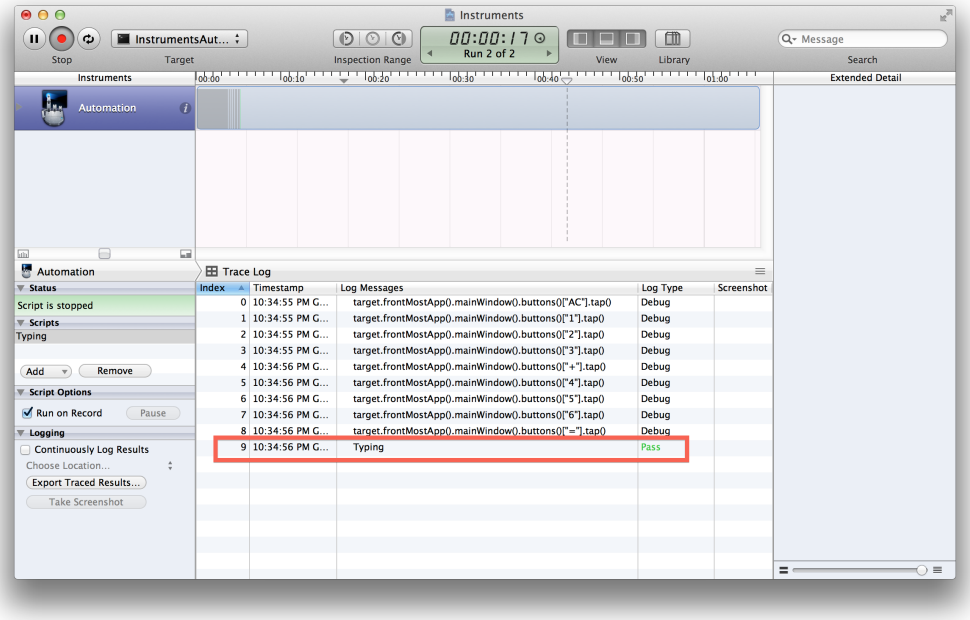

Figure 4.5: Instruments showing a successfully passed test

UI Automation tests can also be created manually, recording the steps required to perform a particular action. Instruments populates the script window with the JavaScript code that performs the equivalent operation. To do this, just click on the "Record" button at the bottom of the screen.

Finally, these tests can also be automated and run from the command line; the syntax for this, however, is quite cumbersome:

```
Line 1  instruments -t /Applications/Xcode.app/Contents/Applications/ ↩
          Instruments.app/Contents/PlugIns/AutomationInstrument. ↩
          bundle/Contents/Resources/Automation.tracetemplate "~/ ↩
          Library/Application Support/iPhone Simulator/6.0/ ↩
          Applications/5B8FC7C7-3384-4A78-867E-D2D07582B5DB/ ↩
          InstrumentsAutomation.app" -e UIASCRIPT Typing.js
```

Pay attention to insert the proper path to the application and to the JavaScript file, in order to be able to execute the test. The output looks like this (edited for brevity):

```
Line 1  Debug: target.frontMostApp().mainWindow().buttons()["AC"].tap()
    -   Debug: target.frontMostApp().mainWindow().buttons()["1"].tap()
    -   Debug: target.frontMostApp().mainWindow().buttons()["2"].tap()
```

```
Debug: target.frontMostApp().mainWindow().buttons()["3"].tap()
Debug: target.frontMostApp().mainWindow().buttons()["+"].tap()
Debug: target.frontMostApp().mainWindow().buttons()["4"].tap()
Debug: target.frontMostApp().mainWindow().buttons()["5"].tap()
Debug: target.frontMostApp().mainWindow().buttons()["6"].tap()
Debug: target.frontMostApp().mainWindow().buttons()["="].tap()
Pass: Typing
Instruments Trace Complete (Duration : 16.224550s; Output :  ↩
    instrumentscli0.trace)
```

4.5 Conclusion

Frank and Calabash represent a quantum leap in terms of UI testing for iOS applications, allowing developers to write testing scenarios in a language close to that of human beings, while at the same time allowing for automated tests of the functionality of a UI.

KIF provides a really complex solution that involves lots of boilerplate code, and requires applications to use the accesibility features of iOS in every application where it is used.

Finally, Instruments allows a greater level of flexibility in the definition of tests, but it uses a JavaScript API that is not very well documented.

Part II

Testing Android Applications

5 Defensive Coding Techniques for Android

This chapter will describe some common techniques, tips and tricks to write high-quality Java code in your Android projects.

5.1 Exceptions

Java has a well-defined exception handling mechanism, but it takes some time to learn to use it effectively. Its misuse can cause trouble to both users and other members of the team.

Types of Exceptions

Java has two categories of exceptions: checked and unchecked. The difference between both is that code advertising checked exceptions in the method signature must be wrapped in `try / catch / finally` statements by developers. Unchecked exceptions represents errors which are not recoverable.

Exception Hierarchy

Figure 5.1 shows the hierarchy of exception classes in Java. All errors and exceptions inherit from `Throwable`; errors are unchecked, as are instances of `RuntimeException`. Developers are expected to recover from checked exceptions safely.

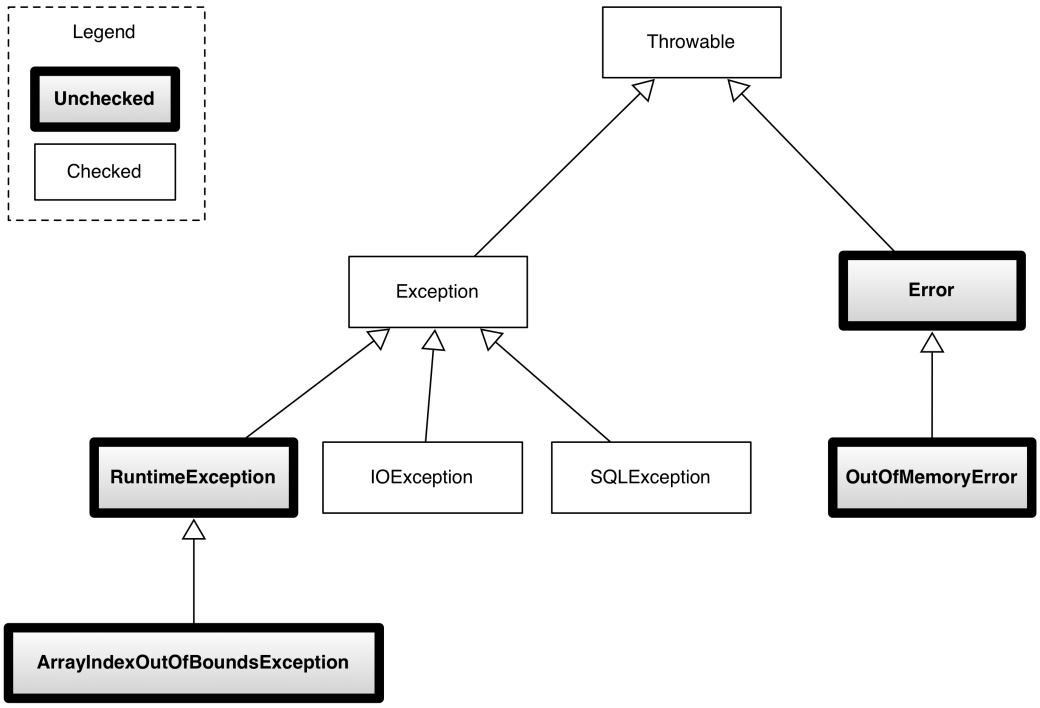

Figure 5.1: Exception class hierarchy in Java

Note

Java and Objective-C use similar names for two opposite concepts; `NSException` is similar to `Error` (it represents non-recoverable situations), while `NSError` is similar to `Exception` (it indicates a recoverable error).

Exception Handling Guidelines

To avoid having code sprinkled with a myriadd of `try / catch / finally` statements, it is recommended to follow the following best practices:

- Catch exceptions as close to the user as possible.
- Code that is meant for reuse (libraries or shared code among multiple applications) should not try to do error handling. It can, however, translate technology-specific exceptions (usually checked) into unchecked, generic ones; as a canonical example, API code could wrap a `FileNotFoundException` (checked) into a `RuntimeException` that would ultimately be thrown.
- Always report exceptions, and report only once. Do not leave empty `catch` blocks in your code, and do not rethrow after doing something with the ex-

ception (like logging). Particularly in Android, exceptional situations might be interesting for the end user, and should be reported.

- Prefer toasts for unimportant information, and only use dialogs for important notifications that require the attention of the user.

5.2 Assertions

Use the `assert` keyword in your programs to test for particular situations during development, and run production code with assertions turned off:

```
Line 1  public int divide(int a, int b) throws IllegalArgumentException {
    -       assert b != 0 : "The second parameter should not be zero!";
    -
    -       if (b == 0) {
    5           this.setLastResult(0);
    -           throw new IllegalArgumentException("Argument 'b' is 0");
    -       }
    -
    -       this.setLastResult(a / b);
    10      return this.getLastResult();
    -   }
```

Even with assertions off, the code documents the fact that `b` is not expected to be zero in this context.

5.3 The Monkey

The Android Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

The Monkey is a command-line tool that that you can run on any emulator instance or on a device. It sends a pseudo-random stream of user events into the system, which acts as a stress test on the application software you are developing.

The Monkey includes a number of options, but they break down into four primary categories:

- Basic configuration options, such as setting the number of events to attempt.

- Operational constraints, such as restricting the test to a single package.
- Event types and frequencies.
- Debugging options.

When the Monkey runs, it generates events and sends them to the system. It also watches the system under test and looks for three conditions, which it treats specially:

- If you have constrained the Monkey to run in one or more specific packages, it watches for attempts to navigate to any other packages, and blocks them.
- If your application crashes or receives any sort of unhandled exception, the Monkey will stop and report the error.
- If your application generates an application not responding error, the Monkey will stop and report the error.

Depending on the verbosity level you have selected, you will also see reports on the progress of the Monkey and the events being generated.

The monkey can be launched from the command line:

```
Line 1  #!/usr/bin/env sh
        -
        -  # This utility runs the Android Monkey
        -
        5  # Variable pointing to the location of the Android SDK
        -  ANDROID_SDK=/Applications/Android/sdk
        -  LOG=monkey.log
        -
        -  if [ -d "$LOG" ]; then
       10      rm $LOG
        -  fi
        -
        -  $ANDROID_SDK/platform-tools/adb shell monkey -p com.akosma.calculator ↩
               --throttle 100 -s 43686 -v 50000 | tee $LOG
```

The parameters of the `adb shell monkey` command are the following:

- `-p` specifies the package name to test.
- `--throttle` specifies the delay in milliseconds between the events.
- `-s` specifies a seed value for the random number generator. This value should be changed every so often, to generate different interactions on the application UI.

- `-v` specifies the verbose option.
- `50000` is the number of events to be simulated.

The Monkey requires an instance of the Emulator running, with the application specified in the `-p` parameter installed in it. The emulator can be run from the command line using the script below:

```
#!/usr/bin/env sh

# This utility runs the Android Emulator

# Variable pointing to the location of the Android SDK
ANDROID_SDK=/Applications/Android/sdk

# Name of the VMD defined in your system
VMD_NAME=Emulator

$ANDROID_SDK/tools/emulator -avd $VMD_NAME
```

The output of the Monkey tool looks like this (edited for brevity):

```
...
:Sending Touch (ACTION_DOWN): 0:(310.0,195.0)
:Sending Touch (ACTION_UP): 0:(398.46033,164.18097)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,2.0)
:Sending Touch (ACTION_DOWN): 0:(785.0,685.0)
:Sending Touch (ACTION_UP): 0:(800.0,763.3364)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-3.0)
:Sending Trackball (ACTION_MOVE): 0:(-1.0,-2.0)
    //[calendar_time:2013-01-29 14:08:36.853  system_uptime:106295]
    // Sending event #100
:Sending Touch (ACTION_DOWN): 0:(274.0,1077.0)
:Sending Touch (ACTION_UP): 0:(270.2971,1078.4357)
:Sending Touch (ACTION_DOWN): 0:(28.0,84.0)
:Sending Touch (ACTION_UP): 0:(11.232578,80.12936)
:Sending Touch (ACTION_DOWN): 0:(547.0,1189.0)
:Sending Touch (ACTION_UP): 0:(629.37836,1216.0)
:Sending Trackball (ACTION_MOVE): 0:(-3.0,-5.0)
:Sending Touch (ACTION_DOWN): 0:(209.0,113.0)
...
```

This log file will report any crashes or anomalies in the execution of the application.

Note

Monkey sometimes causes problems with the adb server. If needed, use the following commands to restart the adb server: `adb kill-server; adb start-server`.

5.4 Performance Tips

The Android documentation features several useful application performance tips that are worth enumerating here:

1. Avoid Creating Unnecessary Objects
2. Prefer Static Over Virtual
3. Use Static Final For Constants
4. Avoid Internal Getters/Setters
5. Use Enhanced For Loop Syntax
6. Consider Package Instead of Private Access with Private Inner Classes
7. Avoid Using Floating-Point
8. Know and Use the Libraries
9. Use Native Methods Carefully
10. Know And Use The Libraries
11. Use Native Methods Judiciously

5.5 Miscellaneous Tips

This section presents a series of simple tips and tricks that can be useful to create quality Android applications.

StrictMode

StrictMode is a developer tool which detects things you might be doing by accident and brings them to your attention so you can fix them.

StrictMode is most commonly used to catch accidental disk or network access on the application's main thread, where UI operations are received and animations take place. Keeping disk and network operations off the main thread

makes for much smoother, more responsive applications. By keeping your application's main thread responsive, you also prevent ANR dialogs from being shown to users.

To enable StrictMode, extend your project adding a custom subclass of the `android.app.Application` class:

```
Line 1  package com.akosma.calculator;
-
-   import android.app.Application;
-   import android.os.Build;
5   import android.os.StrictMode;
-
-   public class CalcApplication extends Application {
-
-           @Override
10          public void onCreate() {
-                   super.onCreate();
-
-                   if (Build.VERSION.SDK_INT >= 9 && isDebug()) {
                        // (1)
Line 14                     StrictMode.enableDefaults();
15
-   //                      StrictMode.setThreadPolicy(new StrictMode.
        ThreadPolicy.Builder()
-   //                                      .detectDiskReads().
        detectDiskWrites().detectNetwork()
-   //                                      .penaltyLog().build());
-   //                      StrictMode.setVmPolicy(new StrictMode.
        VmPolicy.Builder()
20  //                                      .detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects()
-   //                                      .penaltyLog().penaltyDeath().
        build());  (2)
Line 22                 }
-           }
-
25          private boolean isDebug() {
-                   boolean isDebug = ("google_sdk".equals(Build.PRODUCT)
                        )
-                                   || ("sdk".equals(Build.PRODUCT));
-                   return isDebug;
-           }
30  }
```

❶ We only set the `StrictMode` when executing our application in the emulator; this code should not be executed in production applications.

❷ You can also call "detectAll()" for detecting all problems.

To use `android.os.StrictMode`, remember to set your minimum SDK version to 9 in your `AndroidManifest.xml` file. Also, include the application class in the same file, so that it is loaded by the application on startup:

```
Line 1  <?xml version="1.0" encoding="utf-8"?>
-       <manifest xmlns:android="http://schemas.android.com/apk/res/android"
-           package="com.akosma.calculator"
-           android:versionCode="1"
5           android:versionName="1.0" >
-
-           <uses-sdk
-               android:minSdkVersion="9"
-               android:targetSdkVersion="17" />  <!-- ❶ -->
10
-           <application
-               android:allowBackup="true"
-               android:icon="@drawable/ic_launcher"
-               android:label="@string/app_name"
15             android:theme="@style/AppTheme"
-              android:name="com.akosma.calculator.CalcApplication">    <!-- ↩
                   ❷ -->
Line 17            <activity
-                      android:name="com.akosma.calculator.MainActivity"
-                      android:label="@string/app_name" >
20                    <intent-filter>
-                         <action android:name="android.intent.action.MAIN" />
-                         <category android:name="android.intent.category. ↩
                              LAUNCHER" />
-                     </intent-filter>
-                 </activity>
25          </application>
-       </manifest>
```

❶ Here we set the minimum SDK requirement for the application.

❷ Here we specify the name of our custom application class.

Give Threads a Name

Give every thread a meaningful name. This includes thread pool threads. It makes stack dumps much more meaningful. It takes a little more effort to give a meaningful name to even thread pool threads, but if one thread pool has a problem in a long running application, the developer can cause a stack dump to occur, grab the logs, and without having to interrupt a running system you can tell which threads are deadlocked, leaking, growing, etc.

Immutable Objects

The Java API has no concept of immutable objects. The `final` modifier can be used in this case.

For example, if a getter returns a `List` object, make its getter return an immutable view on it, which blocks client code from inadvertently modifying it, using the `Collections.unmodifiableList()` method:

```
Line 1  public List<T> getList() {
    -       return Collections.unmodifiableList(list);
    -   }
```

More

- Sometimes it is useful to use final on local variables to make sure they never change their value. I found this useful in ugly, but necessary loop constructs. Its just to easy to accidently reuse a variable even though it is mend to be a constant.
- Use defense copying in your getters. Unless you return a primitive type or a immutable object make sure you copy the object to not violate encapsulation.
- Never use clone, use a copy constructor.
- Learn the contract between `equals` and `hashCode`. Very often the former is overridden, but not the latter.
- Rather then `var.equals("whatever")` use `"whatever".equals(var)`. That way, if `var == null` there will be no `NullPointerException` thrown.

5.6 Conclusion

This chapter has introduced some useful concepts to write better Android applications.

6 Unit Testing Android Applications

This chapter will introduce basic techniques used to unit test Android applications.

6.1 JUnit

Testing for Android can be classified into tests which only require the JVM and tests which require the Android system.

If you want to run standard JUnit tests on the JVM you cannot access Android classes, because all methods in `android.jar` will throw a `RuntimeException`. This is because `android.jar` does not contain the Android framework code but only stubs for the type signatures, methods, types, etc. The `android.jar` will not be bundled with your application, once you application is deployed on the device it will use the real JAR on the device.

Unfortunately this makes a priori impossible to test Android framework classes on a pure JVM. To test Android classes you need to run them on an Android device or emulator. This unfortunately makes the execution of tests longer. However, the Robolectric framework described in the following section tries to overcome this situation.

JUnit is the standard unit testing framework for Java applications. It is bundled with the official Android SDK distribution.

The preferred way for organizing tests is to create a separate test project for them. There are wizards for creating test projects; then can be found under "File / New / Other" and then select "Android / Android Test Project" in the dialog.

The wizards add the project which should be tested as dependency to the test project. It also create a version of the `AndroidManifest.xml` file which specifies that the `android.test.runner` test library should be used and it specifies an instrumentation.

```
Line 1  <?xml version="1.0" encoding="utf-8"?>
-       <manifest xmlns:android="http://schemas.android.com/apk/res/android"
-           package="com.akosma.calculator.test"
-           android:versionCode="1"
5           android:versionName="1.0" >
-
-           <uses-sdk android:minSdkVersion="8" />
-
-           <instrumentation
10              android:name="android.test.InstrumentationTestRunner"
-               android:targetPackage="com.akosma.calculator" />    (1)
Line 12
-           <application
-               android:icon="@drawable/ic_launcher"
15              android:label="@string/app_name" >
-               <uses-library android:name="android.test.runner" />
-           </application>
-
-       </manifest>
```

(1) This line specifies the package to be tested by the project.

A test project specifies the package of the application to test in the `AndroidManifest.xml` file the under `android:targetPackage` attribute.

Adding Tests

Android supports both JUnit 3 and JUnit 4; in the former case, your test needs to extend the `AndroidTestCase` class, and all test methods must start with the prefix `test`.

Instrumentation allows to control a visible part of the application, e.g. an Activity. For this your testcase would extend `ActivityInstrumentationTestCase2`. This instrumentation class allows you to start and stop activities, run actions on the user interface thread, send key events and more.

To create a new test case, right-click on the package of your test project (under the `src` folder) and select "New / JUnit Test Case". Figure 6.1 shows the corresponding configuration dialog.

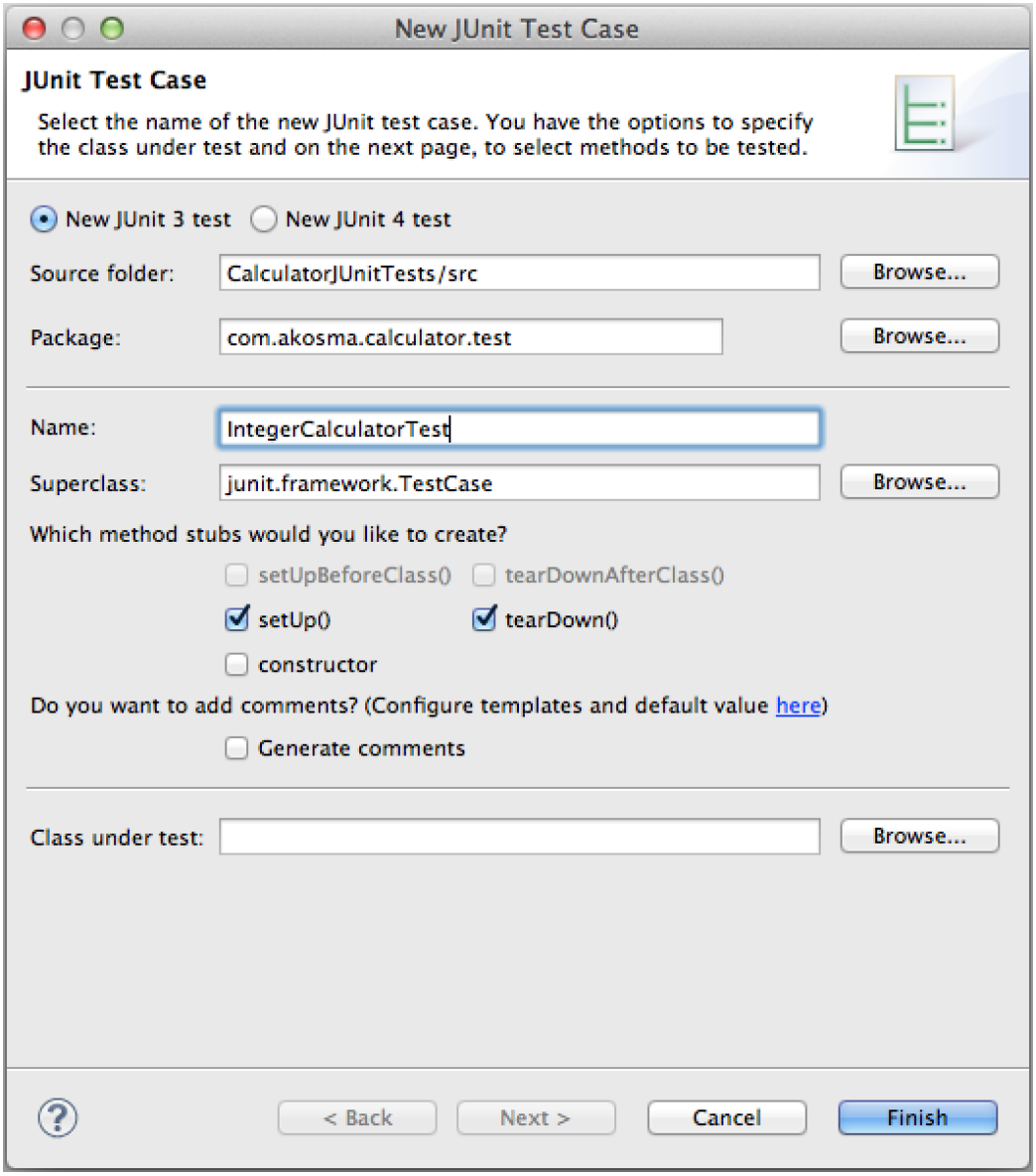

Figure 6.1: Creating a new JUnit test

We are going to add the code required for testing our integer calculator class:

```
Line 1  package com.akosma.calculator.test;
-
-       import android.test.AndroidTestCase;
-       import com.akosma.calculator.IntegerCalculator;
5
-       public class IntegerCalculatorTest extends AndroidTestCase {
```

```
-
-       protected IntegerCalculator calc;
-
10      protected void setUp() throws Exception {
-               super.setUp();
-               this.calc = new IntegerCalculator();
-       }
-
15      protected void tearDown() throws Exception {
-               super.tearDown();
-       }
-
-       public void testCalcShouldNotBeNil() {
20              assertNotNull("The calculator should not be nil",  ↩
                    this.calc);
-       }
-
-       public void testCalcCanAdd() {
-               int result = this.calc.add(3, 5);
25              assertEquals("The result should be the 3 plus 5", 3 + ↩
                     5, result);
-       }
-
-       public void testCalcCanSubstract() {
-               int result = this.calc.substract(3, 5);
30              assertEquals("The result should be the 3 minus 5", 3  ↩
                    - 5, result);
-       }
-
-       public void testCalcCanMultiply() {
-               int result = this.calc.multiply(3, 5);
35              assertEquals("The result should be the 3 times 5", 3  ↩
                    * 5, result);
-       }
-
-       public void testCalcCanDivide() {
-               int result = this.calc.divide(8, 5);
40              assertEquals("The result should be the 8 divided by 5 ↩
                    ", 8 / 5, result);
-       }
-
-       public void testCalcRaisesExceptionIfDivideByZero() {
-               try {
```

```
45                          this.calc.divide(4, 0);
-                           fail("should've thrown an exception!");     // ↩
                                ❶
Line 47                 } catch (IllegalArgumentException e) {
-                           // success!
-                       }
50      }
-   }
```

❶ In JUnit 4 and later, you can use the `@Test` annotation with the `ExpectedException` rule, like this: `@Test(expected = IllegalArgumentException.class)`.

Once this is done, click on the "Run" button on Eclipse, and watch the tests being run. You will see the results in the sidebar, as shown in Figure 6.2.

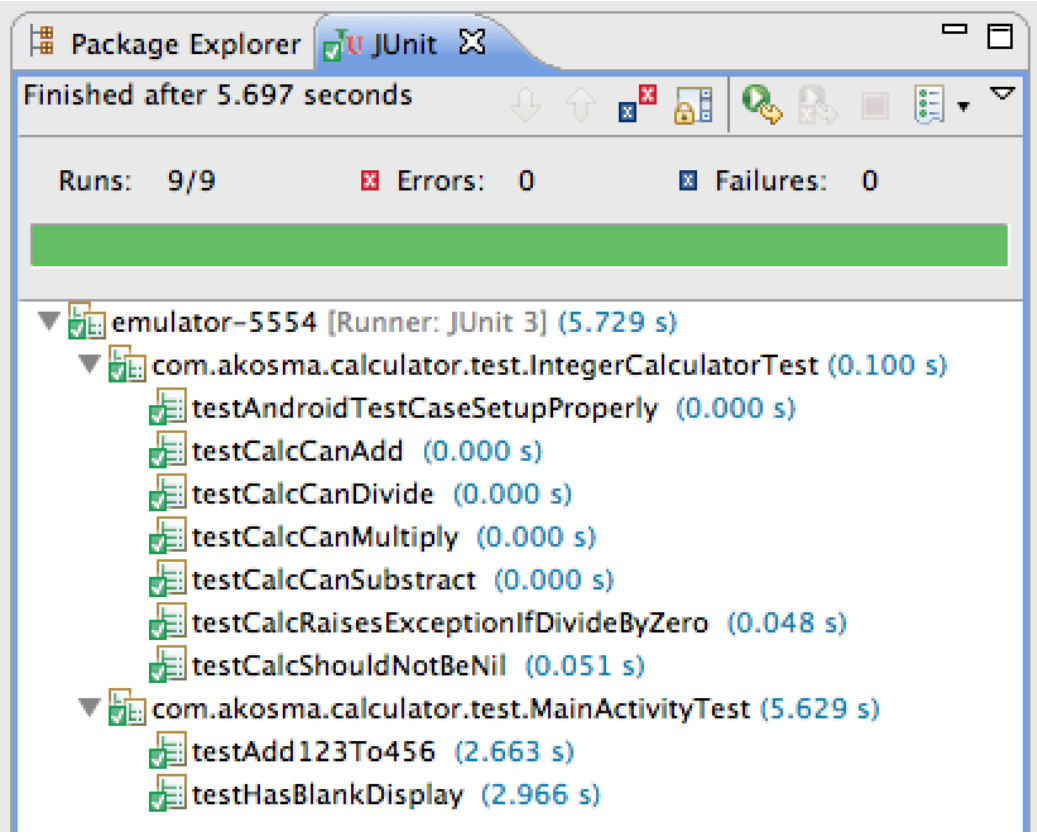

Figure 6.2: JUnit test run results

Testing Activities

You can use this same framework to test your application in a functional way, that is, driving the use of your activity as if a user was playing with the application. We create a subclass of `ActivityInstrumentationTestCase2` for that:

```
Line 1  package com.akosma.calculator.test;
-
-   import android.test.ActivityInstrumentationTestCase2;
-   import android.test.UiThreadTest;
5   import android.widget.Button;
-   import android.widget.GridView;
-   import android.widget.TextView;
-   import com.akosma.calculator.MainActivity;
-   import com.akosma.calculator.R;
10
-   public class MainActivityTest extends  ↩
        ActivityInstrumentationTestCase2<MainActivity> {
-
-       public MainActivityTest() {
-           super(MainActivity.class);
15      }
-
-       public void testHasBlankDisplay() {
-           TextView textView = (TextView) getActivity(). ↩
                findViewById(R.id.txtDisplay);
-           CharSequence text = textView.getText();
20          assertTrue("Upon start, the display should be empty", ↩
                text.equals("0"));
-       }
-
-       @UiThreadTest                          // ❶
Line 24     public void testAdd123To456() {
25          TextView textView = (TextView) getActivity(). ↩
                findViewById(R.id.txtDisplay);
-           GridView keypad = (GridView) getActivity(). ↩
                findViewById(R.id.grdButtons);
-           Button one = (Button)keypad.getChildAt(8);
-           Button two = (Button)keypad.getChildAt(9);
-           Button three = (Button)keypad.getChildAt(10);
30          Button four = (Button)keypad.getChildAt(4);
-           Button five = (Button)keypad.getChildAt(5);
-           Button six = (Button)keypad.getChildAt(6);
-           Button add = (Button)keypad.getChildAt(15);
```

```
                Button equal = (Button)keypad.getChildAt(14);

                one.performClick();
                two.performClick();
                three.performClick();

                CharSequence text = textView.getText();
                assertTrue("The display should say 123", text.equals( ↩
                    "123"));

                add.performClick();

                text = textView.getText();
                assertTrue("The display should say 0", text.equals("0 ↩
                    "));

                four.performClick();
                five.performClick();
                six.performClick();

                text = textView.getText();
                assertTrue("The display should say 456", text.equals( ↩
                    "456"));

                equal.performClick();

                text = textView.getText();
                assertTrue("The display should say 579", text.equals( ↩
                    "579"));
        }
}
```

❶ The `@UiThreadTest` annotation is required here to avoid runtime crashes, because we are calling the `performClick()` method on `Button` instances, and this can only be done in the main UI thread.

The result of this execution is shown in Figure 6.2.

6.2 Robolectric

Robolectric is a unit test framework that mocks the Android SDK jar so that tests run inside the JVM instead of running in the emulator, overcoming the limitation described at the beginning of the previous section.

Installation

The Robolectric documentation states that it requires Maven, but it is actually possible to use it without Maven. Just follow these steps:

1. Create a separate project for testing purposes. Do not create an Android project, or an Android JUnit project. Instead, create a plain old Java Project. (Right click on 'Package Explorer', 'New' > 'Java Project').
2. Name your Robolectric test project something nice, that parallels the main project you are trying to test. Under 'Project layout', choose the default 'Create separate folders for sources and class files'.
3. Hit Next, to get to the Java build settings.
   - On the 'Source' tab, the default 'src' folder is where your test cases will go.
   - On the 'Projects' tab, add your main Android project as a required project on the build path.
   - On the 'Libraries' tab, you will need to add the Android jars, JUnit jars, and the Robolectric jar. Copy the following jar files to the libs folder of your new project, and add them to this tab. In my case, I created the project first, then copied the jars into the libs folder, then returned to this tab in Project settings to add the dependencies.
   - android.jar (from the android sdk root/platforms/android-8 folder)
   - maps.jar (from the Google Play SDK, see note below for information)
   - robolectric-all.jar (downloaded from the Robolectric github page)
   - junit.jar (downloaded from the junit.org website, version 4.10 only!)
4. Hit 'Finish'. Your project will be created.
5. At this point, move into src, then create a test package, and a test case class.
6. Set up Run Configurations for your test Project:
   - From the Run menu, choose 'Run Configurations.' Create a new JUnit test configuration. Do not create an 'Android JUnit Test'. Name your configuration, choose JUnit 4 for the test runner. Make sure 'Run all tests in the selected project, package or source folder' is checked, and choose the name of your test project.
   - At the bottom of the dialog, there may be a prompt, "Multiple launchers available — Select one...". Click the 'Select other...' link, checoject's

working directory to the folder containing your main test project. Select the 'Arguments' tab. At the bottom under 'Working Directory', select the 'Other' radio button. Click the 'Workspace...' button and select the Android project that you are running tests against.

- click 'Run' and your configuration will be saved and your tests run.

Note

Robolectric has a direct dependency on the Google Maps API, so you will need to install it using the Android SDK manager, which is available running the `sdk/tools/android sdk` command line in the root of your Android folder. Figure 6.3 shows which option to select. The procedure is explained in detail in the Setup documentation for the Google Play SDK.
Also, do not use JUnit 4.11, instead use JUnit 4.10.

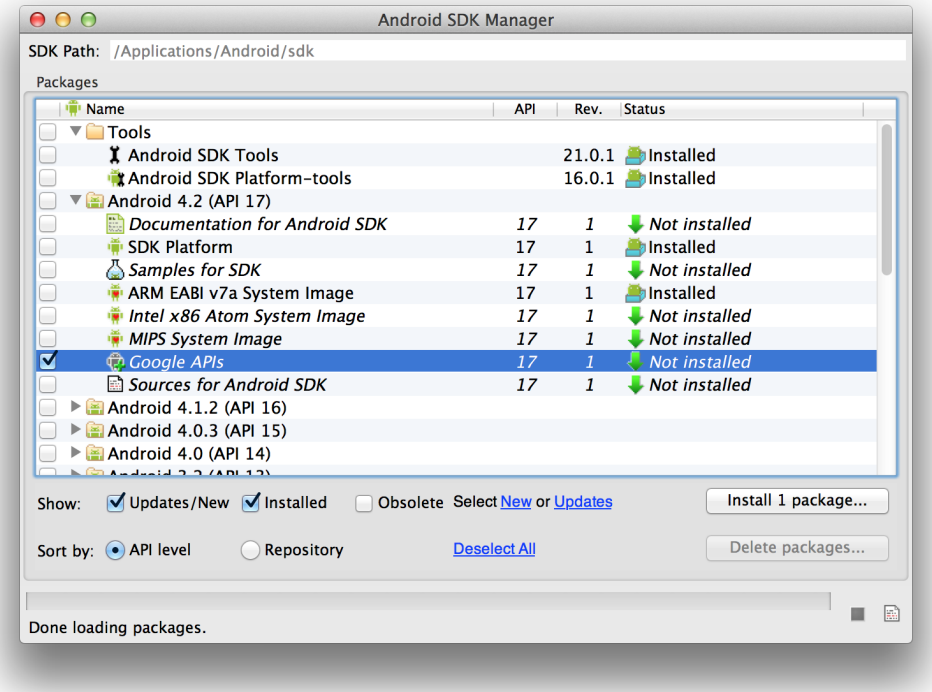

Figure 6.3: Android SDK Manager

Adding Tests

After doing all this, a Robolectric test looks like this:

```
Line 1  package com.akosma.calculator.robolectric;
-
-   import org.junit.After;
-   import org.junit.Before;
5   import org.junit.Test;
-   import org.junit.runner.RunWith;
-
-   import com.akosma.calculator.IntegerCalculator;
-   import com.xtremelabs.robolectric.RobolectricTestRunner;
10
-   import static org.hamcrest.CoreMatchers.equalTo;
-   import static org.junit.Assert.assertNotNull;
-   import static org.junit.Assert.assertThat;
-   import static org.junit.Assert.fail;
15
-   @RunWith(RobolectricTestRunner.class)
-   public class IntegerCalculatorTest {
-
-           protected IntegerCalculator calc;
20
-           @Before
-           public void setUp() throws Exception {
-                   this.calc = new IntegerCalculator();
-           }
25
-           @After
-           public void tearDown() throws Exception {
-           }
-
30         @Test
-           public void testCalcShouldNotBeNil() {
-                   assertNotNull("The calculator should not be nil",  ↩
                        this.calc);
-           }
-
35         @Test
-           public void testCalcCanAdd() {
-                   int result = this.calc.add(3, 5);
-                   assertThat("The result should be the 3 plus 5", 3 +  ↩
                        5, equalTo(result));
-           }
40
```

```
        @Test
        public void testCalcCanSubstract() {
                int result = this.calc.substract(3, 5);
                assertThat("The result should be the 3 minus 5", 3 - 5, equalTo(result));
        }

        @Test
        public void testCalcCanMultiply() {
                int result = this.calc.multiply(3, 5);
                assertThat("The result should be the 3 times 5", 3 * 5, equalTo(result));
        }

        @Test
        public void testCalcCanDivide() {
                int result = this.calc.divide(8, 5);
                assertThat("The result should be the 8 divided by 5", 8 / 5, equalTo(result));
        }

        @Test(expected=IllegalArgumentException.class)
        public void testCalcRaisesExceptionIfDivideByZero() {
                this.calc.divide(4, 0);
                fail("should've thrown an exception!");
    }
}
```

As shown above, the Robolectric library uses JUnit 4 tests.

You can also, up to a certain point, use Robolectric for functional testing:

```
Line 1  package com.akosma.calculator.robolectric;

        import com.xtremelabs.robolectric.RobolectricTestRunner;

        import org.junit.After;
        import org.junit.Before;
        import org.junit.Test;
        import org.junit.runner.RunWith;

        //import android.widget.Button;
        //import android.widget.GridView;
        import android.widget.TextView;
```

```
import android.os.Bundle;

import com.akosma.calculator.MainActivity;
import com.akosma.calculator.R;

//import static com.xtremelabs.robolectric.Robolectric.clickOn;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.junit.Assert.assertThat;

//import static org.junit.Assert.assertTrue;

@RunWith(RobolectricTestRunner.class)
public class MainActivityTest {

        private MainActivity activity;
        private TextView textView;

        @Before
        public void setUp() throws Exception {

                activity = new MainActivity();
                activity.onCreate(new Bundle());
                textView = (TextView) activity.findViewById(R.id. ↩
                    txtDisplay);
        }

        @After
        public void tearDown() throws Exception {
        }

        @Test
        public void testHasBlankDisplay() {
                String text = (String) textView.getText();
                assertThat("Upon start, the display should be empty", ↩
                     text,
                                equalTo("0"));
        }

        @Test
        public void testAdd123To456() {

                // Unfortunately Robolectric does not appear
                // to work with GridViews + AdaptorsÉ
```

```
                // This code remains in suspension for the moment.

                // GridView keypad = (GridView) activity.findViewById ↩
                    (R.id.grdButtons);
                // Button one = (Button) keypad.getChildAt(8);
                // Button two = (Button) keypad.getChildAt(9);
                // Button three = (Button) keypad.getChildAt(10);
                // Button four = (Button) keypad.getChildAt(4);
                // Button five = (Button) keypad.getChildAt(5);
                // Button six = (Button) keypad.getChildAt(6);
                // Button add = (Button) keypad.getChildAt(15);
                // Button equal = (Button) keypad.getChildAt(14);

                // clickOn(one);
                // clickOn(two);
                // clickOn(three);
                //
                // CharSequence text = textView.getText();
                // assertTrue("The display should say 123", text. ↩
                    equals("123"));
                //
                // clickOn(add);
                //
                // text = textView.getText();
                // assertTrue("The display should say 0", text.equals ↩
                    ("0"));
                //
                // clickOn(four);
                // clickOn(five);
                // clickOn(six);
                //
                // text = textView.getText();
                // assertTrue("The display should say 456", text. ↩
                    equals("456"));
                //
                // clickOn(equal);
                //
                // text = textView.getText();
                // assertTrue("The display should say 579", text. ↩
                    equals("579"));
        }
}
```

Unfortunately, the support of Robolectric for `GridView` instances is broken at the time of this writing, and it is impossible to retrieve pointers to the children `Button` instances that are set through the `KeypadAdapter` adapter.

6.3 Conclusion

This chapter has introduced some techniques that can be used to unit test Android applications effectively, using both JUnit and Robolectric.

7 Functional Testing for Android Apps

This chapter will showcase two different techniques to add functional tests to Android applications: Calabash-Android and Robotium.

7.1 Calabash-Android

We have previously talked about Calabash in Chapter 4, when we introduced Frank and Calabash-iOS. The Android version is also built on top of Cucumber and offers a very similar workflow to the two tools used for iOS apps.

However, the current version of Calabash-Android is under heavy development, and the instructions in the paragraphs below might not work in the future.

Note

At the time of this writing, the latest version of Calabash-Android is version 0.3.8.

Getting Started

The first thing we are going to do is to install the required libraries; just like in the case of Calabash-iOS, we are going to use RubyGems for that:

```
Line 1  gem install calabash-android
```

Once this is done, cd to the folder where your Android application is located, and type this command:

```
Line 1  calabash-android gen
```

This will generate the required folder structure, containing a sample features file.

Preparing the Android Project

At the time of this writing, the current version of Calabash-Android requires a few changes in the Android project to run successfully; otherwise, brace for some errors when running the tests.

1. Make sure that the project is set to use version 7 of the Android SDK as a minimum.
2. Add the `android.permission.INTERNET` permission to your application.

Both settings must be specified in the `AndroidManifest.xml` file:

```
Line 1  <?xml version="1.0" encoding="utf-8"?>
-       <manifest xmlns:android="http://schemas.android.com/apk/res/android"
-           package="com.akosma.calculator"
-           android:versionCode="1"
5           android:versionName="1.0" >
-
-           <uses-sdk
-               android:minSdkVersion="7" />     (1)
Line 9
10          <application
-               android:allowBackup="true"
-               android:icon="@drawable/ic_launcher"
-               android:label="@string/app_name"
-               android:theme="@style/AppTheme" >
15              <activity
-                   android:name="com.akosma.calculator.MainActivity"
-                   android:label="@string/app_name" >
-                   <intent-filter>
-                       <action android:name="android.intent.action.MAIN" />
20
-                       <category android:name="android.intent.category. ↩
                            LAUNCHER" />
-                   </intent-filter>
-               </activity>
-           </application>
25
-           <uses-permission android:name="android.permission.INTERNET" /> (2)
Line 27
-       </manifest>
```

❶ Minimum version of the Android SDK supported by the application being tested.

❷ Permission to access the internet.

Once you have made these changes, make sure to clean the project build in Eclipse, and rebuild the project.

Creating a Feature

You can just rename or remove the autogenerated feature, and then create a new one with the following contents:

```
Line 1  Feature: Typing numbers
-
-       Scenario: Calculate
-         Then I press the "1" button
5         Then I press the "2" button
-         Then I press the "3" button
-         Then I press the " \+ " button
-         Then I press the "4" button
-         Then I press the "5" button
10        Then I press the "6" button
-         Then I press the "=" button
-         Then I should see "579"
```

The default feature syntax is not exactly the same as the one for Calabash-iOS; however, you can make both systems speak the same language by defining custom steps. You can see the default list of steps bundled by Calabash-Android in the project wiki on Github.

Running the Test

Calabash-Android tests are run using the following command

```
Line 1  calabash-android run bin/CalabashTests.apk
```

The `bin/CalabashTests.apk` is the path of the APK file to test.

Warning
Please make sure that the Android Emulator is running before you run `calabash-android run <APK_PATH>`.

The command above will run the application in the Android Emulator, executing each of the steps, one by one. The output of the command should look like this (edited for brevity):

```
Feature: Typing numbers

  Scenario: Calculate              # features/typing.feature:3
833 KB/s (434707 bytes in 0.509s)
858 KB/s (182718 bytes in 0.207s)
    Then I press the "1" button
    Then I press the "2" button
    Then I press the "3" button
    Then I press the " \+ " button
    Then I press the "4" button
    Then I press the "5" button
    Then I press the "6" button
    Then I press the "=" button
    Then I should see "579"

1 scenario (1 passed)
9 steps (9 passed)
1m25.243s
```

7.2 Robotium

Robotium is an open source functional testing library for Android applications. It is very simple to set up and use, and allows to create tests using JUnit that are executed on the emulator, simulating user actions.

These are the advantages of Robotium:

- You can develop powerful test cases, with minimal knowledge of the application under test.
- The framework handles multiple Android activities automatically.
- Minimal time needed to write solid test cases.
- Readability of test cases is greatly improved, compared to standard instrumentation tests.
- Test cases are more robust due to the run-time binding to GUI components.
- Blazing fast test case execution.
- Integrates smoothly with Maven or Ant to run tests as part of continuous integration.

Note
At the time of this writing, the latest version of Robotium is 3.6.

How to use

To use Robotium, follow these steps:

- Create a project of type "Android Test Project".
- Add a JUnit Test Case file to the root package of your project, and select `ActivityInstrumentationTestCase2` as the superclass.
- Add the latest Robotium jar in the `libs` folder of your project, and include it in the "Java Build Path" properties ("Libraries" tab) or the properties of your project.
- Add the code for your test:

```
Line 1  package com.akosma.calculator.test;
-
-   import android.test.ActivityInstrumentationTestCase2;
-   import android.widget.TextView;
5   import com.jayway.android.robotium.solo.Solo;
-   import com.akosma.calculator.MainActivity;
-   import com.akosma.calculator.R;
-
-   public class FunctionalTest extends ActivityInstrumentationTestCase2< ↩
        MainActivity> {
10      private Solo solo;
-
-       public FunctionalTest() {
-               super(MainActivity.class);
-       }
15
-       protected void setUp() throws Exception {
-               super.setUp();
-               solo = new Solo(getInstrumentation(), getActivity());
-       }
20
-       protected void tearDown() throws Exception {
-               super.tearDown();
-       }
-
25      public void testHasBlankDisplay() {
```

```
                TextView textView = (TextView) solo.getView(R.id.
                    txtDisplay);
                CharSequence text = textView.getText();
                assertTrue("Upon start, the display should be empty",
                     text.equals("0"));
        }

        public void testAdd123To456() {
                solo.clickOnButton("1");
                solo.clickOnButton("2");
                solo.clickOnButton("3");

                TextView textView = (TextView) solo.getView(R.id.
                    txtDisplay);
                CharSequence text = textView.getText();
                assertTrue("Display should show 123", text.equals("
                    123"));

                solo.clickOnButton(" \\+ ");
                text = textView.getText();
                assertTrue("Display should show 0", text.equals("0"))
                    ;

                solo.clickOnButton("4");
                solo.clickOnButton("5");
                solo.clickOnButton("6");
                text = textView.getText();
                assertTrue("Display should show 456", text.equals("
                    456"));

                solo.clickOnButton("=");
                text = textView.getText();
                assertTrue("Display should show 579", text.equals("
                    579"));
        }
}
```

Finally, right-click on the `FunctionalTest.java` file and choose "Run As / Android JUnit Test". The emulator will launch and the results of the test will be shown, as usual, in the JUnit pane in Eclipse.

7.3 Conclusion

Calabash-Android provides a mechanism based on the popular Cucumber framework, and offers a set of functionality very similar to that of Cucumber-iOS, becoming a very interesting choice for teams working in cross-platform mobile development.

On the other hand, Robotium offers a handy way to extend JUnit tests to perform functional testing using a very simple API, with an extremely simple setup.

Colophon

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 2009-2013 akosma software. All rights reserved. Written in Switzerland. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, or discounts for bulk purchases and special sales, please contact Adrian Kosmaczewski (+ 41 78 739 47 76, books@akosma.com)

Visit our website at akosma.com.

Created using Asciidoc. Kindle version generated with KindleGen by Amazon. Text typed on Vim and MacVim 7.3 on Mac OS X 10.8.2.

First edition, January 2013.

Bibliography

Books

[1] [sommerville] Ian Sommerville. Software Engineering, Eigth Edition. Addison Wesley. 2007. ISBN 978-0-321-31379-9.

[2] [mcconnell] Steve McConnell. Code Complete, Second Edition. Microsoft Press. 2004. ISBN 0-7356-1967-0.

[3] [richardson] Jared Richardson & William Gwaltney Jr. Ship It!: A Practical Guide to Successful Software Projects. The Pragmatic Programmers. 2005. ISBN 978-0-9745-1404-8.

[4] [gunderloy] Mike Gunderloy. Coder to Developer. Sybex. 2004. ISBN 0-7821-4327-X.

[5] [meyers] Scott Meyers. Effective C++, Third Edition. Addison-Wesley. 2005. ISBN 978-0-321-33487-9.

[6] [lee] Graham Lee. Test-Driven iOS Development. Addison-Wesley. 2012. ISBN 978-0-321-77418-7.

[7] [steinberg] Daniel Steinberg. Test Driving iOS Development with Kiwi. Dim Sum Thinking, Inc. 2012. ISBN 978-0-9830669-0-3.

Tools

[8] ACRA

[9] Android Tools Download

[10] Cedar

[11] CocoaPods

[12] FindBugs

[13] GHUnit

[14] Google Toolbox for Mac

[15] HockeyKit

[16] KIF

[17] Kiwi

[18] Roboelectric

[19] Specta

[20] XcodeCoverage

[21] JUnit

[22] The Cocotron

[23] GNUStep

[24] The LLVM Compiler Infrastructure

[25] clang: a C language family frontend for LLVM

[26] Clang Static Analyzer

[27] Robotium

[28] AndroVM

Articles

[29] iOS Debugging Magic

[30] Improved logging in Objective-C

[31] Understanding and Analyzing iOS Application Crash Reports

[32] UI/Application Exerciser Monkey

[33] Activity Testing Tutorial

[34] Performance Tips

[35] Code Style Guidelines for Contributors

[36] Android Application Error Reports

[37] Favorite (Clever) Defensive Programming Best Practices

[38] How do I set up NSZombieEnabled in Xcode 4?

[39] Objective-C: Assertion vs. Exception vs. Error

[40] How do you implement global iPhone Exception Handling?

[41] "Debug certificate expired" error in Eclipse Android plugins

[42] Getting a useful stack trace from NSException#callStackReturnAddresses

[43] Writing Your First Frank Test

[44] Debugging Core Data Objects

[45] TDD vs BDD

[46] OCUnit

[47] Kiwi Library

[48] Introduction to Android development : TouchCalculator

[49] Tracking Down Crashes with Exception Breakpoints

[50] A few Android tips

[51] Android Testing with the Android Test framework, Robotium, Monkey and Robolectric

[52] Be careful when using JUnit's expected exceptions

[53] Android Unit Testing in Robolectric, using Eclipse

[54] Calabash-Android Canned Steps

[55] Let's Build NSObject

[56] Practical Blocks

[57] Network Link Conditioner in Lion

A iOS Coding Guidelines

To improve readability and maintainability of your Objective-C code, we recommend following these coding standards:

A.1 Files, Code Organization and Other Issues

- Do not include more than one class in the same file. Use one file per class, with the filename matching the name of the class, for easier retrieval.
- Put protocols in a separate file. Do not include protocols in class headers.
- Lines are indented with 4 spaces, which means **NO TABS**.
- Add a space between numbers, mathematical operators and other symbols.
- Class names use PascalCase, variables and methods use camelCase, constants use `ALL_CAPS_WITH_UNDERSCORES`.
- Unless they are assigned some value in the declaration, always initialize all pointer variables to nil, no matter whether they are instance fields or not.
- Use the `@class` statement in header files (for both classes and protocols) whenever possible, instead of `#import`'ing them, to reduce the number of dependencies between files.
- Use ARC (Automatic Reference Counting) for new projects.

A.2 Brackets

- The layout of brackets follows the "Allman" or "ANSI style", with opening bracket in a separate line:

```
Line 1  @interface SomeClass : NSObject
-       {
-           // ivars go here
-       }
```

- Always use opening brackets, even in one-line blocks, such as the following if and while statements:

```
Line 1  while(x == y)
-       {
-           [something message];
-       }
5       [other message];
-
-       if(x == y)
-       {
-           [something message];
10      }
```

A.3 Instance Variable + Property Naming Standards

- Against Apple's own recommendation, **ivars must have an underscore as prefix**.
- Properties have the same name as the ivars they wrap, minus the underscore:

```
Line 1  @interface SomeClass : NSObject
-       {
-           NSInteger _ivar;
-       }
5
-       @property (nonatomic) NSInteger ivar;
-
-       @end
```

- Do not override @synthesized properties with your own code: use @dynamic properties instead, **particularly when you are not wrapping an existing ivar**.
- Always use accessors to get or set ivars; do not access them directly. This will make your code KVO- and KVC-compliant, and in Objective-C this is important.
- Append the "IBOutlet" decoration to properties, not to ivars:

```
Line 1  @interface SomeClass : NSObject
    -   {
    -       UILabel *_ivar;
    -   }
    5
    -   @property (nonatomic, retain) IBOutlet UILabel *ivar;
    -
    -   @end
```

A.4 Properties, init and dealloc

- To avoid raising KVO notifications during init and dealloc, do not ever use setters or getters in those two methods.
- In all other methods, always use the setters and getters, to be sure to raise KVO notifications.
- Use the `self.prop = nil;` idiom for releasing a retained object and to set its wrapped ivar to nil at the same time. This syntax is the same for retained and assigned properties.

```
Line 1  - (id)init
    -   {
    -       if (self = [super init])
    -       {
    5           _prop = [[SomeClass alloc] init];
    -       }
    -       return self;
    -   }
    -
    10  - (id)someMethod:(id)param
    -   {
    -       // ...
    -
    -       // This generates KVO notifications, sets the ivar to nil and ↩
                properly
    15      // releases the object as required.
    -       self.prop = nil;
    -
    -       // ...
    -
    20      return nil;
```

```
}
```

A.5 Pointers

- Pointer variables always feature a space between the class name and the star sign. The star sign and the ivar variable name are not separated by a space:

```
- (ReturnClass *)methodName:(ParamClass1 *)param1 another:( ↩
    ParamClass2 *)param2
{
    SomeOtherClass *variable = [[SomeOtherClass alloc] init];
    return nil;
}
```

A.6 Comments

- Self-documenting code is a myth; always add comments to explain what's going on. More comments is better.
- Add HeaderDocs to class and method definitions. These comments can later be extracted using Doxygen or HeaderDoc:

```
/*!
    @class       SomeController
    @superclass  UIViewController
    @abstract    Does interesting things, indeed.
*/
@interface SomeController : UIViewController
{
}

/*!
    @method     doSomething
    @abstract   Changes the internal state of the object.
    @discussion Unbelievable things can happen if you call this ↩
        method.
    @result     Just a float number meaning lots of things.
*/
- (float)doSomething;
```

A.7 Protocols

- Albeit Objective-C is a dynamic language, **do not use informal protocols**; create explicit files with protocol definitions.
- Put protocol definitions in their own source code files, never in the same file of the class which uses the protocol methods, neither in the class used as delegate.
- Follow Apple's guidelines for naming the methods of a protocol; in particular, make sure that the first parameter of every protocol method is a pointer to the object calling the method:

```
Line 1  [source,c]
-       @protocol SomeControllerDelegate
-
-       - (void)someController:(SomeController *)controller ↩
            didSomethingWithThisObject:(id)object;
```

- Use the @required and @optional keywords to explicitly separate methods of a protocol which **must** be implemented from those that **should** be implemented:

```
Line 1  @protocol SomeControllerDelegate
-
-       @required
-       - (void)someController:(SomeController *)controller ↩
            didSomethingWithThisObject:(id)object;
5
-       @optional
-       - (void)someControllerDidSomethingElse:(SomeController *)controller;
-
-       @end
```

A.8 Before Committing Code in SCM Systems

- Do not leave compiler warnings unattended. **Objective-C code must always compile without warnings – warnings must be treated as errors.**
- Use the LLVM/Clang Static Analyzer to check your code for memory leaks or unforeseen problems (integrated in Xcode 3.2). Set the "Run Static Analyzer" option for the Debug configuration of your project to make the static analyzer run automatically at each build.
- Run all unit tests.

- Always add a meaningful message with your commit.

Code Style Guidelines for Android

These guidelines have been adapted from the Code Style Guidelines for Android Contributors.

B.1 Java Language Rules

We follow standard Java coding conventions. We add a few rules:

Don't Ignore Exceptions

Sometimes it is tempting to write code that completely ignores an exception like this:

```
Line 1  void setServerPort(String value) {
-           try {
-               serverPort = Integer.parseInt(value);
-           } catch (NumberFormatException e) { }
5   }
```

You must never do this. While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day. You must handle every Exception in your code in some principled way. The specific handling varies depending on the case.

Anytime somebody has an empty catch clause they should have a creepy feeling. There are definitely times when it is actually the correct thing to do, but at least you have to think about it. In Java you can't escape the creepy feeling. -James Gosling

Acceptable alternatives (in order of preference) are:

- Throw the exception up to the caller of your method.

```
Line 1  void setServerPort(String value) throws NumberFormatException {
    -       serverPort = Integer.parseInt(value);
    -   }
```

- Throw a new exception that's appropriate to your level of abstraction.

```
Line 1  void setServerPort(String value) throws ConfigurationException {
    -       try {
    -           serverPort = Integer.parseInt(value);
    -       } catch (NumberFormatException e) {
    5           throw new ConfigurationException("Port " + value + " is not ↩
                    valid.");
    -       }
    -   }
```

- Handle the error gracefully and substitute an appropriate value in the catch {} block.

```
Line 1  /** Set port. If value is not a valid number, 80 is substituted. */
    -
    -   void setServerPort(String value) {
    -       try {
    5           serverPort = Integer.parseInt(value);
    -       } catch (NumberFormatException e) {
    -           serverPort = 80;  // default port for server
    -       }
    -   }
```

- Catch the Exception and throw a new `RuntimeException`. This is dangerous: only do it if you are positive that if this error occurs, the appropriate thing to do is crash.

```
Line 1  /** Set port. If value is not a valid number, die. */
    -
    -   void setServerPort(String value) {
    -       try {
    5           serverPort = Integer.parseInt(value);
    -       } catch (NumberFormatException e) {
    -           throw new RuntimeException("port " + value " is invalid, ", ↩
                    e);
    -       }
    -   }
```

Note that the original exception is passed to the constructor for RuntimeException. If your code must compile under Java 1.3, you will need to omit the exception that is the cause. * Last resort: if you are confident that actually ignoring the exception is appropriate then you may ignore it, but you must also comment why with a good reason:

```
Line 1  /** If value is not a valid number, original port number is used. ↩
          */
        void setServerPort(String value) {
            try {
                serverPort = Integer.parseInt(value);
        5   } catch (NumberFormatException e) {
                // Method is documented to just ignore invalid user input.
                // serverPort will just be unchanged.
            }
        }
```

Don't Catch Generic Exception

Sometimes it is tempting to be lazy when catching exceptions and do something like this:

```
Line 1  try {
            someComplicatedIOFunction();        // may throw IOException
            someComplicatedParsingFunction();   // may throw ParsingException
            someComplicatedSecurityFunction();  // may throw  ↩
                SecurityException
        5   // phew, made it all the way
        } catch (Exception e) {                 // I'll just catch all  ↩
            exceptions
            handleError();                      // with one generic handler!
        }
```

You should not do this. In almost all cases it is inappropriate to catch generic Exception or Throwable, preferably not Throwable, because it includes Error exceptions as well. It is very dangerous. It means that Exceptions you never expected (including RuntimeExceptions like ClassCastException) end up getting caught in application-level error handling. It obscures the failure handling properties of your code. It means if someone adds a new type of Exception in the code you're calling, the compiler won't help you realize you need to handle that error differently. And in most cases you shouldn't be handling different types of exception the same way, anyway.

There are rare exceptions to this rule: certain test code and top-level code where you want to catch all kinds of errors (to prevent them from showing up in a UI, or to keep a batch job running). In that case you may catch generic Exception (or Throwable) and handle the error appropriately. You should think very carefully before doing this, though, and put in comments explaining why it is safe in this place.

Alternatives to catching generic Exception:

- Catch each exception separately as separate catch blocks after a single try. This can be awkward but is still preferable to catching all Exceptions. Beware repeating too much code in the catch blocks.
- Refactor your code to have more fine-grained error handling, with multiple try blocks. Split up the IO from the parsing, handle errors separately in each case.
- Rethrow the exception. Many times you don't need to catch the exception at this level anyway, just let the method throw it.

Remember: exceptions are your friend! When the compiler complains you're not catching an exception, don't scowl. Smile: the compiler just made it easier for you to catch runtime problems in your code.

Don't Use Finalizers

Finalizers are a way to have a chunk of code executed when an object is garbage collected.

Pros: can be handy for doing cleanup, particularly of external resources.

Cons: there are no guarantees as to when a finalizer will be called, or even that it will be called at all.

Decision: we don't use finalizers. In most cases, you can do what you need from a finalizer with good exception handling. If you absolutely need it, define a close() method (or the like) and document exactly when that method needs to be called. See InputStream for an example. In this case it is appropriate but not required to print a short log message from the finalizer, as long as it is not expected to flood the logs.

Fully Qualify Imports

When you want to use class Bar from package foo,there are two possible ways to import it:

1. `import foo.*;`

- Pros: Potentially reduces the number of import statements.

2. `import foo.Bar;`

   - Pros: Makes it obvious what classes are actually used. Makes code more readable for maintainers.

Decision: Use the latter for importing all Android code. An explicit exception is made for java standard libraries (`java.util.*`, `java.io.*`, etc.) and unit test code (`junit.framework.*`)

B.2 Java Library Rules

There are conventions for using Android's Java libraries and tools. In some cases, the convention has changed in important ways and older code might use a deprecated pattern or library. When working with such code, it's okay to continue the existing style (see Consistency). When creating new components never use deprecated libraries.

B.3 Java Style Rules

Use Javadoc Standard Comments

Files should start with a package statement and import statements should follow, each block separated by a blank line. And then there is the class or interface declaration. In the Javadoc comments, describe what the class or interface does.

```
Line 1  package com.android.internal.foo;
-
-       import android.os.Blah;
-       import android.view.Yada;
5
-       import java.sql.ResultSet;
-       import java.sql.SQLException;
-
-       /**
10      * Does X and Y and provides an abstraction for Z.
-       */
-
-       public class Foo {
-           ...
15      }
```

Every class and nontrivial public method you write *must* contain a Javadoc comment with at least one sentence describing what the class or method does. This sentence should start with a 3rd person descriptive verb.

Examples:

```
Line 1  /** Returns the correctly rounded positive square root of a double ↩
            value. */
        static double sqrt(double a) {
            ...
        }
```

or

```
Line 1  /**
         * Constructs a new String by converting the specified array of
         * bytes using the platform's default character encoding.
         */
        public String(byte[] bytes) {
            ...
        }
```

You do not need to write Javadoc for trivial get and set methods such as `setFoo()` if all your Javadoc would say is "sets Foo". If the method does something more complex (such as enforcing a constraint or having an important side effect), then you must document it. And if it's not obvious what the property "Foo" means, you should document it.

Every method you write, whether public or otherwise, would benefit from Javadoc. Public methods are part of an API and therefore require Javadoc.

Android does not currently enforce a specific style for writing Javadoc comments, but you should follow the Sun Javadoc conventions.

Define Fields in Standard Places

Fields should be defined either at the top of the file, or immediately before the methods that use them.

Limit Variable Scope

The scope of local variables should be kept to a minimum (*Effective Java* Item 29). By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

One exception to this rule concerns try-catch statements. If a variable is initialized with the return value of a method that throws a checked exception, it must be initialized inside a try block. If the value must be used outside of the try block, then it must be declared before the try block, where it cannot yet be sensibly initialized:

```
Line 1  // Instantiate class cl, which represents some sort of Set
    -   Set s = null;
    -   try {
    -       s = (Set) cl.newInstance();
    5   } catch(IllegalAccessException e) {
    -       throw new IllegalArgumentException(cl + " not accessible");
    -   } catch(InstantiationException e) {
    -       throw new IllegalArgumentException(cl + " not instantiable");
    -   }
   10
    -   // Exercise the set
    -   s.addAll(Arrays.asList(args));
```

But even this case can be avoided by encapsulating the try-catch block in a method:

```
Line 1  Set createSet(Class cl) {
    -       // Instantiate class cl, which represents some sort of Set
    -       try {
    -           return (Set) cl.newInstance();
    5       } catch(IllegalAccessException e) {
    -           throw new IllegalArgumentException(cl + " not accessible");
    -       } catch(InstantiationException e) {
    -           throw new IllegalArgumentException(cl + " not instantiable");
    -       }
   10  }
    -
    -   ...
    -
    -   // Exercise the set
   15   Set s = createSet(cl);
    -   s.addAll(Arrays.asList(args));
```

Loop variables should be declared in the for statement itself unless there is a compelling reason to do otherwise:

```
Line 1  for (int i = 0; i n; i++) {
    -       doSomething(i);
    -   }
```

and

```
Line 1  for (Iterator i = c.iterator(); i.hasNext(); ) {
    -       doSomethingElse(i.next());
    -   }
```

Order Import Statements

The ordering of import statements is:

1. Android imports
2. Imports from third parties (`com`, `junit`, `net`, `org`)
3. `java` and `javax`

To exactly match the IDE settings, the imports should be:

- Alphabetical within each grouping, with capital letters before lower case letters (e.g. Z before a).
- There should be a blank line between each major grouping (`android`, `com`, `junit`, `net`, `org`, `java`, `javax`).

Originally there was no style requirement on the ordering. This meant that the IDE's were either always changing the ordering, or IDE developers had to disable the automatic import management features and maintain the imports by hand. This was deemed bad. When java-style was asked, the preferred styles were all over the map. It pretty much came down to our needing to "pick an ordering and be consistent." So we chose a style, updated the style guide, and made the IDEs obey it. We expect that as IDE users work on the code, the imports in all of the packages will end up matching this pattern without any extra engineering effort.

This style was chosen such that:

- The imports people want to look at first tend to be at the top (`android`)
- The imports people want to look at least tend to be at the bottom (`java`)
- Humans can easily follow the style
- IDEs can follow the style

The use and location of static imports have been mildly controversial issues. Some people would prefer static imports to be interspersed with the remaining imports, some would prefer them reside above or below all other imports. Additinally, we have not yet come up with a way to make all IDEs use the same ordering.

Since most people consider this a low priority issue, just use your judgement and please be consistent.

Use Spaces for Indentation

We use 4 space indents for blocks. We never use tabs. When in doubt, be consistent with code around you.

We use 8 space indents for line wraps, including function calls and assignments. For example, this is correct:

```
Instrument i =
        someLongExpression(that, wouldNotFit, on, one, line);
```

and this is not correct:

```
Instrument i =
    someLongExpression(that, wouldNotFit, on, one, line);
```

Follow Field Naming Conventions

- Non-public, non-static field names start with m.
- Static field names start with s.
- Other fields start with a lower case letter.
- Public static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

For example:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

Use Standard Brace Style

Braces do not go on their own line; they go on the same line as the code before them. So:

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

We require braces around the statements for a conditional. Except, if the entire conditional (the condition and the body) fit on one line, you may (but are not obligated to) put it all on one line. That is, this is legal:

```
if (condition) {
    body();
}
```

and this is legal:

```
if (condition) body();
```

but this is still illegal:

```
if (condition)
    body();  // bad!
```

Limit Line Length

Each line of text in your code should be at most 100 characters long.

There has been lots of discussion about this rule and the decision remains that 100 characters is the maximum.

Exception: if a comment line contains an example command or a literal URL longer than 100 characters, that line may be longer than 100 characters for ease of cut and paste.

Exception: import lines can go over the limit because humans rarely see them. This also simplifies tool writing.

Use Standard Java Annotations

Annotations should precede other modifiers for the same language element. Simple marker annotations (e.g. @Override) can be listed on the same line with the language element. If there are multiple annotations, or parameterized annotations, they should each be listed one-per-line in alphabetical order.<

Android standard practices for the three predefined annotations in Java are:

- `@Deprecated`: The @Deprecated annotation must be used whenever the use of the annotated element is discouraged. If you use the @Deprecated annotation, you must also have a @deprecated Javadoc tag and it should name an alternate implementation. In addition, remember that a @Deprecated method is *still supposed to work.*

  If you see old code that has a @deprecated Javadoc tag, please add the @Deprecated annotation. * `@Override`: The @Override annotation must be used whenever a method overrides the declaration or implementation from a superclass.

  For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method. * `@SuppressWarnings`: The @SuppressWarnings annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation *must* be used, so as to ensure that all warnings reflect actual problems in the code.

  When a @SuppressWarnings annotation is necessary, it must be prefixed with a TODO comment that explains the "impossible to eliminate" condition. This will normally identify an offending class that has an awkward interface. For example:

```
Line 1  // TODO: The third-party class com.third.useful.Utility.rotate() ↩
            needs generics
    -   @SuppressWarnings("generic-cast") List<String> blix = Utility. ↩
            rotate(blax);
```

  When a @SuppressWarnings annotation is required, the code should be refactored to isolate the software elements where the annotation applies.

Treat Acronyms as Words

Treat acronyms and abbreviations as words in naming variables, methods, and classes. The names are much more readable:

Good	**Bad**
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID
class Html	class HTML
String url	String URL
long id	long ID

Both the JDK and the Android code bases are very inconsistent with regards to acronyms, therefore, it is virtually impossible to be consistent with the code around you. Bite the bullet, and treat acronyms as words.

For further justifications of this style rule, see *Effective Java* Item 38 and *Java Puzzlers* Number 68.

Log Sparingly

While logging is necessary it has a significantly negative impact on performance and quickly loses its usefulness if it's not kept reasonably terse. The logging facilities provides five different levels of logging. Below are the different levels and when and how they should be used.

- `ERROR`: This level of logging should be used when something fatal has happened, i.e. something that will have user-visible consequences and won't be recoverable without explicitly deleting some data, uninstalling applications, wiping the data partitions or reflashing the entire phone (or worse). This level is always logged. Issues that justify some logging at the ERROR level are typically good candidates to be reported to a statistics-gathering server.
- `WARNING`: This level of logging should used when something serious and unexpected happened, i.e. something that will have user-visible consequences but is likely to be recoverable without data loss by performing some explicit action, ranging from waiting or restarting an app all the way to re-downloading a new version of an application or rebooting the device. This level is always logged. Issues that justify some logging at the WARNING level might also be considered for reporting to a statistics-gathering server.
- `INFORMATIVE`: This level of logging should used be to note that something interesting to most people happened, i.e. when a situation is detected that is likely to have widespread impact, though isn't necessarily an error. Such a condition should only be logged by a module that reasonably believes that it is the most authoritative in that domain (to avoid duplicate logging by non-authoritative components). This level is always logged.

- `DEBUG`: This level of logging should be used to further note what is happening on the device that could be relevant to investigate and debug unexpected behaviors. You should log only what is needed to gather enough information about what is going on about your component. If your debug logs are dominating the log then you probably should be using verbose logging.

  This level will be logged, even on release builds, and is required to be surrounded by an `if (LOCAL_LOG)` or `if (LOCAL_LOGD)` block, where `LOCAL_LOG[D]` is defined in your class or subcomponent, so that there can exist a possibility to disable all such logging. There must therefore be no active logic in an `if (LOCAL_LOG)` block. All the string building for the log also needs to be placed inside the `if (LOCAL_LOG)` block. The logging call should not be re-factored out into a method call if it is going to cause the string building to take place outside of the `if (LOCAL_LOG)` block.

  There is some code that still says `if (localLOGV)`. This is considered acceptable as well, although the name is nonstandard. * `VERBOSE`: This level of logging should be used for everything else. This level will only be logged on debug builds and should be surrounded by an `if (LOCAL_LOGV)` block (or equivalent) so that it can be compiled out by default. Any string building will be stripped out of release builds and needs to appear inside the `if (LOCAL_LOGV)` block.

Notes:

- Within a given module, other than at the VERBOSE level, an error should only be reported once if possible: within a single chain of function calls within a module, only the innermost function should return the error, and callers in the same module should only add some logging if that significantly helps to isolate the issue.
- In a chain of modules, other than at the VERBOSE level, when a lower-level module detects invalid data coming from a higher-level module, the lower-level module should only log this situation to the DEBUG log, and only if logging provides information that is not otherwise available to the caller. Specifically, there is no need to log situations where an exception is thrown (the exception should contain all the relevant information), or where the only information being logged is contained in an error code. This is especially important in the interaction between the framework and applications, and conditions caused by third-party applications that are properly handled by the framework should not trigger logging higher than the DEBUG level. The only situations that should trigger logging at the INFORMATIVE level or higher is when a module or application detects an error at its own level or coming from a lower level.

- When a condition that would normally justify some logging is likely to occur many times, it can be a good idea to implement some rate-limiting mechanism to prevent overflowing the logs with many duplicate copies of the same (or very similar) information.
- Losses of network connectivity are considered common and fully expected and should not be logged gratuitously. A loss of network connectivity that has consequences within an app should be logged at the DEBUG or VERBOSE level (depending on whether the consequences are serious enough and unexpected enough to be logged in a release build).
- A full filesystem on a filesystem that is acceessible to or on behalf of third-party applications should not be logged at a level higher than INFORMATIVE.
- Invalid data coming from any untrusted source (including any file on shared storage, or data coming through just about any network connections) is considered expected and should not trigger any logging at a level higher then DEBUG when it's detected to be invalid (and even then logging should be as limited as possible).
- Keep in mind that the `+` operator, when used on Strings, implicitly creates a `StringBuilder` with the default buffer size (16 characters) and potentially quite a few other temporary String objects, i.e. that explicitly creating StringBuilders isn't more expensive than relying on the default + operator (and can be a lot more efficient in fact). Also keep in mind that code that calls `Log.v()` is compiled and executed on release builds, including building the strings, even if the logs aren't being read.
- Any logging that is meant to be read by other people and to be available in release builds should be terse without being cryptic, and should be reasonably understandable. This includes all logging up to the DEBUG level.
- When possible, logging should be kept on a single line if it makes sense. Line lengths up to 80 or 100 characters are perfectly acceptable, while lengths longer than about 130 or 160 characters (including the length of the tag) should be avoided if possible.
- Logging that reports successes should never be used at levels higher than VERBOSE.
- Temporary logging that is used to diagnose an issue that's hard to reproduce should be kept at the DEBUG or VERBOSE level, and should be enclosed by if blocks that allow to disable it entirely at compile-time.
- Be careful about security leaks through the log. Private information should be avoided. Information about protected content must definitely be avoided. This

is especially important when writing framework code as it's not easy to know in advance what will and will not be private information or protected content.

- `System.out.println()` (or `printf()` for native code) should never be used. System.out and System.err get redirected to /dev/null, so your print statements will have no visible effects. However, all the string building that happens for these calls still gets executed.
- *The golden rule of logging is that your logs may not unnecessarily push other logs out of the buffer, just as others may not push out yours.*

Be Consistent

Our parting thought: BE CONSISTENT. If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding, so people can concentrate on what you're saying, rather than on how you're saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Try to avoid this.

B.4 Javatests Style Rules

Follow Test Method Naming Conventions

When naming test methods, you can use an underscore to seperate what is being tested from the specific case being tested. This style makes it easier to see exactly what cases are being tested.

For example:

```
Line 1  testMethod_specificCase1 testMethod_specificCase2
-
-       void testIsDistinguishable_protanopia() {
-           ColorMatcher colorMatcher = new ColorMatcher(PROTANOPIA)
5           assertFalse(colorMatcher.isDistinguishable(Color.RED, Color.BLACK ↩
                ))
-           assertTrue(colorMatcher.isDistinguishable(Color.X, Color.Y))
-       }
```