

Xcode Tools Sensei

Your Guide to the Mac OS X and iOS
Developer Tools

Mark Szymczyk

Me and Mark Publishing

Published by Me and Mark Publishing
<http://www.meandmark.com>

Copyright ©2012 by Mark Szymczyk

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means without written permission from the publisher or author.

TRADEMARKS — Apple, Mac, Macintosh, Mac OS X, Xcode, Cocoa, Cocoa Touch, AppleScript, App Store, Finder, iPad, iPhone, iPod, iTunes, QuickTime, and Safari are trademarks of Apple Computer, Inc. PowerPC is a trademark of International Business Machines Corporation. OpenGL is a trademark of Silicon Graphics, Inc. All other products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations.

DISCLAIMER — This book was independently published by Me and Mark Publishing. Apple is not affiliated with this book. Apple does not endorse or support this book.

Printed in the United States of America

Table of Contents

Acknowledgments 21

Introduction 22

Getting Xcode 4	22
Accessing Other Developer Tools	23
The Book's Contents	23
Xcode Preferences	24
What the Reader Needs to Know	25
Some Things to Keep in Mind as You Read This Book	25

Chapter 1: Xcode Projects 26

Creating a Project	26
Step 1: Choose the Type of Project You Want to Create	26
Step 2: Choose a Product Name	27
Step 3: Save the Project	28
The Project's Contents	28
Application Projects	29
Cocoa Projects	29
Document and Non-Document Applications	30
Core Data Applications	30
Command-Line Tool Projects	30
Framework and Library Projects	31
Libraries	31
Frameworks	31
Bundles and XPC Services	31
Application Plug-in Projects	32
System Plug-in Projects	32
Other Projects	33
iOS Application Projects	34
Devices	35
Core Data	35
Storyboarding	35
iOS Library Projects	35
Project Window	36
Toolbar	36
Navigator	37
Project Navigator	38
Project Navigator Groups	38
Symbol Navigator	39
Filtering the Symbol Navigator's Contents	40
Search Navigator	40

4 Table of Contents

Customizing Your Search.....	41
Find Scopes	41
Find and Replace	42
Issue Navigator.....	43
Debug Navigator	43
Breakpoint Navigator	43
Log Navigator.....	43
Editor.....	44
Utility Area	44
File Inspector.....	44
Identity and Type	45
Localization	45
Target Membership	45
Text Settings	46
Source Control	46
Quick Help Inspector	46
Library.....	46
Debug Area	47
Adding Files and Frameworks to Your Project	47
Creating New Files for the Project.....	47
Choosing a File Type	48
Naming the File.....	48
Mac File Types.....	49
Cocoa	49
C and C++	50
User Interface	50
Core Data	50
Resource.....	50
Other	51
iOS File Types.....	51
Cocoa Touch	51
C and C++	52
User Interface	53
Core Data	53
Resource.....	53
Other	54
Fixing the Copyright Notice	54
Adding Files You've Already Created	55
Adding a Folder of Files.....	56
Adding Files to Targets.....	56
Adding Frameworks and Libraries to a Project	56
Source Trees.....	57
Removing Files from a Project.....	57
Renaming a Project.....	58
Modernizing a Project.....	58

Workspaces	58
Creating a Workspace	59
Adding Projects to a Workspace	59
Organizer.....	59
Opening the Organizer.....	59
Organizer for iOS Applications	60
Developer Profile	61
Provisioning Profiles.....	61
Software Images	61
Device Logs	62
Screenshots	62
Devices	62

Chapter 2: Editing Source Code 64

The Editor Pane	64
Jump Bar.....	64
Editor and Gutter	65
Focus Ribbon.....	65
Assistant Editor.....	66
Code Completion.....	67
Customizing Code Editing.....	67
Fonts and Colors Preferences	68
Color Themes	68
Categories	68
Setting Colors for Non-Text Items	68
Text Editing Preferences	68
Indentation Preferences	69
Tabs.....	69
Line Wrapping.....	70
Syntax-Aware Indenting.....	70
Key Bindings.....	70
Code Snippets.....	71
Using a Code Snippet	71
Creating a Code Snippet	72
Completion Shortcuts	72
Completion Scope	72
Placing Tokens in Your Code Snippets	73
Examining a Code Snippet	74
Tab Bar.....	74
Refactoring Tools.....	74
Converting Your Project to ARC.....	75
Converting to Modern Objective-C Syntax.....	76
Fix-it	77
Reading Developer Documentation	77

6 Table of Contents

Browsing Documentation	78
Searching Documentation	79
Bookmarks.....	80
Quick Help.....	80
Invoking Quick Help	80
What Quick Help Displays.....	81
Updating Documentation	81
Installing Third-Party Documentation.....	82
Removing Documentation Sets.....	82

Chapter 3: Creating User Interfaces for Mac Applications 83

Starting with Interface Builder	83
Creating the User Interface.....	84
Modifying the Interface	84
Selecting an Element.....	85
Selecting an Element in a Hierarchy.....	85
Moving and Resizing Elements	85
Deleting an Element	86
Changing the Text of Titles and Labels	86
Making Other Modifications.....	86
Making Connections	86
Testing the Interface	87
Creating a Xib File.....	87
Object List	87
File's Owner	87
First Responder	88
Application.....	88
Object Library.....	89
Controls	89
Buttons.....	89
Text Controls	90
Miscellaneous Controls	90
Formatters	91
Data Views	92
Cells	93
Layout Views.....	93
Objects and Controllers.....	94
Windows and Menus.....	95
Windows.....	95
Menus	96
Toolbar.....	97
Address Book.....	97
Automator	97
Disc Recorder.....	98

Image Kit	98
OSAKit	99
PDFKit	99
UIKit.....	99
Quartz Composer	99
SceneKit	100
WebKit	100
Custom Objects	100
Media Library.....	100
Inspectors.....	101
File Inspector	101
Quick Help Inspector	102
Identity Inspector	102
Custom Class.....	102
Identity.....	102
Tool Tip	103
Accessibility Identity	103
User Defined Runtime Attributes	103
Document.....	104
Attributes Inspector.....	104
Size Inspector	105
Sizing Controls	105
Setting an Element's Size and Position	105
Autosizing	105
Springs.....	106
Struts.....	106
Aligning Elements.....	106
Positioning Items in a Containing View	107
Sizing Windows	107
Connections Inspector	108
Bindings Inspector	108
View Effects Inspector	109
Turning on Core Animation Effects	109
Appearance Section	109
Content Filters	109
Background Filters.....	110
Compositing Filters.....	110
Transitions for Subviews	110
Working with Menus.....	111
Adding Menus to the Menu Bar	111
Adding Items to a Menu.....	111
Keyboard Equivalents	112
Adding Submenus	112
Creating Contextual Menus.....	112
Creating Dock Menus	113

8 Table of Contents

Attaching Menus to Buttons	113
Bindings.....	114
Creating the Model Class	114
Creating the Controller	114
Binding the Model to the Controller.....	115
Binding the View to the Controller	115
Value Transformers.....	116
Connecting to Your Classes.....	116
Grouping Objects	118
Creating a Matrix of Controls	118
Setting Tab Order	118
Toolbars	119
Adding a Toolbar	119
Image and Custom View Toolbar Items.....	119
Adding Items to the Toolbar	119
Adding Images and Labels to Toolbar Items.....	120
Split Views	120
Adding and Removing Views	121
Arranging and Sizing Subviews.....	121
Embedding a Split View	121
Dividers.....	121
Source Lists	122
Auto Layout.....	122
Turning on Auto Layout.....	122
Constraints.....	123
Editing Constraints.....	123
Content Priorities.....	124
Adding Constraints	125

Chapter 4: Creating User Interfaces for iOS Applications 126

Starting with Interface Builder	126
Creating the User Interface.....	127
Modifying the Interface	128
Selecting an Element.....	128
Selecting an Element in a Hierarchy.....	128
Moving and Resizing Elements	128
Deleting an Element	128
Changing the Text of Titles and Labels	128
Making Other Modifications	129
Making Connections	129
Testing the User Interface	129
Creating a Xib File.....	130
Object List	130
File's Owner	130

First Responder	131
Object Library	131
Controls	131
Data Views	132
Gesture Recognizers	133
Objects and Controllers	133
Windows and Bars	133
Media Library	134
Inspectors	134
File Inspector	135
Quick Help Inspector	135
Identity Inspector	136
Custom Class	136
Identity	136
User Defined Runtime Attributes	136
Document	137
Accessibility	137
Attributes Inspector	138
Size Inspector	138
Setting an Element's Size and Position	138
Autosizing	139
Springs	139
Struts	139
Aligning Elements	140
Positioning Items in a Containing View	140
Sizing Windows	140
Connections Inspector	141
Connecting to Your Classes	141
Grouping Objects	143
Storyboarding	143
Creating a Storyboard	143
The Storyboard Canvas	144
Working with Scenes and Segues	144
Creating a Table in Interface Builder	145
Auto Layout	145
Turning on Auto Layout	145
Constraints	146
Editing Constraints	146
Content Priorities	147
Adding Constraints	147

Chapter 5: Modeling Tools 149

Data Models	149
Adding a Data Model File to Your Project	149
XML Data Models.....	150
Data Model Editor	150
Top-Level Components.....	151
Detail Area	151
Bottom Area	151
Graph View	152
Adding Entities.....	152
Adding Attributes.....	153
Setting an Attribute's Name and Data Type	153
Setting Additional Attribute Information.....	154
Adding Relationships.....	154
Adding Fetched Properties	156
Adding Fetch Requests.....	156
Editing the Fetch Request's Predicate with the Predicate Builder	156
Data Model Inspector for Fetch Requests.....	157
Advanced Checkboxes	158
Setting Information Dictionary Entries.....	158
Adding Configurations.....	159
Versioning	159
Advanced Checkboxes for Attributes and Relationships	160
Synchronizing Data Models.....	160
Syncing an Entity.....	160
Syncing an Attribute	161
Syncing a Relationship	161
Creating Source Code.....	162
Mapping Models	163
Versioned and Non-Versioned Data Models	163
Adding a New Version of Your Data Model	163
Adding a Mapping Model to Your Project.....	164
Mapping Model Editor.....	165
Entity Mappings.....	165
Property Mappings	166
Changing Attribute Mapping Data	166
Changing Relationship Mapping Data	167
Creating a User Dictionary	167
Migrating the Data.....	167
Enabling Automatic Migration	167
Migrating a Document-Based Application	168
Migrating a Regular Application	168

Chapter 6: Building Projects 169

Project Editor	169
Targets.....	169
Inspecting and Configuring Target Settings	170
Summary.....	170
Mac Target Summary.....	170
iOS Target Summary.....	172
Version and Build Numbers	174
Adding an Icon to Your Application.....	174
Info	176
Custom Target Properties.....	176
Document Types	177
iOS Document Types	177
Exported and Imported UTIs	178
URL Types.....	179
Services	180
Build Settings	180
Target Build Phases.....	181
Target Dependencies.....	181
Adding Build Phases	182
Reordering Build Phases	183
Build Rules.....	183
Adding Targets	183
Aggregate Targets.....	184
Unit Testing Bundles.....	184
Adding a Target Dependency	184
Configuring the Unit Test Bundle.....	185
Adding Unit Testing Classes	185
Writing and Running Unit Tests.....	186
Project Settings.....	186
Deployment Target.....	187
What Should My Deployment Target Be?.....	187
Deployment Targets and SDKs.....	188
When Should You Use an Earlier SDK?	188
Build Configurations	189
Localizations	190
Xcode Build Settings.....	190
Architectures	191
Build Locations	193
Build Options.....	193
Picking a Compiler.....	194
Code Signing.....	194
Code Signing iOS Applications	195
Code Signing Mac Applications	195
Creating a Code Signing Identity	196

Code Signing Build Settings	196
Deployment	197
Deployment Target.....	197
Targeted Device Family.....	197
Stripping Symbols	197
Kernel Module	198
Linking.....	198
Packaging.....	199
Search Paths	199
Unit Testing	199
Versioning.....	200
Code Generation.....	200
Optimization Level	201
Generate Debug Symbols	201
Language.....	201
Choosing the Language Compiler	201
Choosing the Language Standard	202
Enabling Exception Handling.....	202
Setting Compiler Flags	202
Enabling Objective-C Automatic Reference Counting	202
Preprocessing.....	203
Warnings.....	203
Data Model Version Compiler	204
Interface Builder Compiler	204
Static Analyzer	204
Conditional Build Settings	205
Adding Your Own Build Settings.....	205
Configuration Settings Files	206
Creating a Configuration Settings File	206
What Goes in a Configuration Settings File?.....	207
Telling Your Project to Use a Configuration Settings File.....	208
Overriding the Configuration Settings File.....	208
Compiling Your Program	208
Schemes	208
Choosing a Scheme.....	209
Opening the Scheme Editor.....	209
Build.....	210
Run.....	210
Test.....	211
Profile	211
Analyze	211
Archive	212
Pre and Post-Actions	212
Adding and Managing Schemes.....	212
Precompiled Headers	213

Cleaning Targets.....	214
Building Your Project	214
Where's My Application?	215
Seeing More Build Details	216
Message Bubbles.....	216
Opening the Build Results Window	216
Showing the Build Transcript.....	217
Filtering the Build Results.....	217
Customizing Xcode Behaviors	218
Tips for Correcting Build Errors	218
Add All Necessary Frameworks	218
Include Necessary Header Files.....	219
The Error May Not Be Where Xcode Says It Is.....	219
One Error Can Cause Multiple Syntax Errors.....	219
Look for Typographical Errors	220
Check Function Arguments.....	220
Building for Unsupported Languages	220
Static Analysis.....	221
Generating Output Files	222
Creating Applications that Run on iPhones and iPads	223
Creating a New Universal Project	223
Upgrading an Existing iPhone Project	223
Universal Application Build Settings	223
Creating Two Device-Specific Applications	224

Chapter 7: Debugging 225

Before You Debug	225
Configuring Your Scheme for Debugging.....	225
Info	225
Arguments.....	226
Options.....	226
Diagnostics	227
Memory Management.....	227
Logging.....	227
Debugger.....	228
Setting Environment Variables for Debugging.....	228
Choosing a Debugging Format.....	230
Breakpoints.....	231
Setting Breakpoints	231
Breakpoint Actions	232
Debugger Command	233
Log.....	233
Sound.....	233
Shell Command.....	233

14 Table of Contents

AppleScript	234
Capture OpenGL ES Frame.....	234
Sharing Breakpoints	234
Launching the Debugger	234
Opening a Separate Console Window	235
Debug Bar	235
Debug Navigator	236
Floating Debugger Window	237
Variables View	237
Setting Watchpoints.....	238
Custom Data Formatters	239
Datatips.....	240
Using Datatips	240
Using Step Controls in the Editor.....	240
Viewing Shared Libraries.....	241
Tracking Expressions.....	241
Viewing Dynamic Arrays	242
Stepping Through Your Code	242
Viewing Memory	243
OpenGL ES Debugging	245
Enabling OpenGL ES Frame Capture.....	245
Capturing the Frame When Reaching a Breakpoint	245
Capturing the Frame	245
Framebuffer Area	246
Debug Navigator	247
Variables View	247
Assistant Editor	248
Labeling OpenGL ES Objects in the Debugger.....	249
Using the GDB Console	249
Stopping Program Execution	249
Setting Breakpoints.....	250
Setting Watchpoints.....	251
Setting Catchpoints.....	251
Examining Your Breakpoints	251
Setting Conditional Breakpoints.....	251
Disabling and Deleting Breakpoints	253
Command Lists.....	255
Examining Data	256
Examining Dynamic Arrays	256
Displaying Data Automatically	257
Executing Shell Commands.....	259
Defining Your Own Commands	259
Conditional Commands.....	260
Documenting Your Commands	261
Reading Commands from a File.....	262

Command Hooks	263
Using the LLDB Console	264
Getting Help.....	264
Setting Breakpoints	265
Setting Watchpoints	265
Examining Breakpoints	266
Disabling and Deleting Breakpoints.....	267
Breakpoint Commands	268
Command Aliases.....	269
Examining Variables.....	269
Examining Memory.....	271
Executing Shell Commands.....	271
LLDB Expressions	273
Logging.....	274

Chapter 8: Version Control 275

Creating a Repository	276
Creating a Local git Repository	276
Creating a Local Subversion Repository	276
Creating a Remote git Repository	276
Creating a Remote Subversion Repository	277
How Many Repositories Should You Make?	277
Ignoring Files	277
Naming the Ignore File	278
What Files Should Be Ignored?	278
What to Do with the Ignore File?	278
Configuring the Repository for Xcode.....	279
Cloning Repositories	280
Repositories Window	280
Repository List	281
Detail View.....	281
History.....	282
Importing Your Project to the Repository.....	282
Importing to a git Repository.....	282
Importing to a Subversion Repository.....	283
Checking Out Files from a Subversion Repository	284
Seeing Which Files Have Changed in Your Project.....	284
Adding Files to the Repository	285
Removing Files from the Repository	286
Seeing the Changes You Made to a File	286
Committing Changes You Made.....	287
Discarding Changes	288
Viewing Annotations	288
Viewing a File's Revisions	289

16 Table of Contents

Branching	289
Creating a Branch	290
Removing a Branch.....	290
Switching Branches.....	290
Merging	291
Tracking Branches.....	291
Pushing and Pulling	292
Snapshots.....	292
Taking a Snapshot.....	293
Looking at a Project's Snapshots	293
Restoring a Snapshot.....	293
Deleting Snapshots	294
Accessing Your Snapshots	294

Chapter 9: Instruments 295

Tracing From Xcode	295
Creating and Setting up a Trace Document.....	296
Creating a Trace Document.....	296
Mac OS X Templates	296
iOS Templates	297
iOS Simulator Templates.....	298
Trace Document Window	298
Adding and Removing Instruments	299
Customizing the Track Pane	299
Showing the Detail View.....	300
Running a Trace	300
Determining What to Trace	300
Choosing a Program to Trace	301
Tracing	301
Recording Options.....	302
Alternate Trace Document Views	302
Examining Trace Results	302
Track Pane	303
Detail View.....	303
Console	303
Source View.....	304
Searching in the Detail View	305
Extended Detail View.....	305
Filtering Information	306
Filtering by Time.....	306
Searching	306
Flagging Samples.....	307
Call Tree Data Mining	307
Call Tree Checkboxes.....	308

Call Tree Constraints	309
Specific Data Mining	309
Data Mining Inside the Call Tree	310
Call Tree Tips.....	310
Run Browser	311
Exporting Trace Data.....	311
Instrument-Specific Results.....	311
Leaks	311
Before You Trace.....	312
Leaked Blocks	312
History	313
Call Tree.....	313
Cycles and Roots.....	314
Track Pane.....	315
Allocations	315
Before You Trace.....	315
Statistics	316
Object Summary	316
Instances	317
History	318
Call Trees	318
Objects List	319
Heapshots	319
Time Profiler	321
Call Tree.....	321
Finding Heavy Paths	322
Focusing on a Subtree	322
Finding Where a Function Spends Its Time	323
Sample List.....	323
Strategy Bar	324
OpenGL ES Analyzer	325
Expert.....	325
Frame Statistics.....	326
Trace	327
Call Trees	327
API Statistics.....	328
Single Frame Navigation.....	328
Overriding the Pipeline	328
Activity Monitor.....	329
Summary	329
Parent Child	330
Samples.....	330
Trace Highlights	331
Creating a Custom Instrument	331
Parts of an Instrument	331

18 Table of Contents

Parts of a Probe	332
Determining When the Probe Fires	332
Performing the Action	334
Custom Instrument Example	334
Start	334
DATA	335
BEGIN	335
First Probe	335
Second Probe	336
Running a Trace	336
Improvements to the Custom Instrument	337
Editing an Instrument	337

Chapter 10: Command-Line Debugging Tools 338

A Command Line Primer	338
Executing Commands as root	338
Navigating Directories	338
Getting Help	340
Finding Your Application's Process ID	340
fs_usage	340
Running fs_usage	341
What fs_usage Tells You	341
fs_usage Options	343
-e Option	343
-f Option	344
-w Option	344
sc_usage	345
What sc_usage Tells You	345
Program Summary Information	345
System Call List	346
sc_usage Options	347
-c Option	347
-e Option	348
-E Option	348
-l Option	348
-s Option	349
vmmap	349
What vmmap Tells You	349
Non-Writable Memory Regions	349
Region Purpose	350
Permissions	351
Sharing Modes	351
Writable Memory Regions	352
Summary Report	353

vmmap Options.....	353
-d Option	354
-w Option	354
-resident Option	354
-pages Option	355
-interleaved Option	355
-submap Option	355
-allSplitLibs Option	355
-noCoalesce Option	356
-v Option	356
heap	356
heap's Output	356
heap Options.....	357
-guessNonObjects Option	358
-sumObjectFields Option	358
-showSizes Option	358
-addresses Option	358
leaks	359
Running leaks	359
What leaks Tells You.....	359
leaks Options.....	360
-nocontext Option.....	360
-nostacks Option	360
-exclude Option	361
malloc_history	361
Running malloc_history	361
Running malloc_history on a Specific Memory Area	362
Showing All Allocation Events.....	362

Chapter 11: OpenGL Tools 363

OpenGL Profiler.....	363
Choosing a Program to Profile	364
Custom Pixel Formats	364
Setting Environment Variables.....	365
Remote Profiling.....	365
Setting Breakpoints	366
Multithread Control	367
Breakpoint Actions	367
Profiling Your Program	367
Viewing the Profiling Data	368
Trace Window	368
Statistics Window	369
Buffers Window.....	369
Resources Window.....	370

Textures and Renderbuffers	370
Programs and Shaders	371
Scripts Window	371
Breakpoints Window	372
Pixel Format Window	372
Messages Window	372
OpenGL Driver Monitor	373
Getting Started	373
Customizing the Graph	374
Table View	374
Renderer Info	375
OpenGL Shader Builder	375
Creating a Project	376
Adding Shaders	377
Writing a Shader	378
Adding Textures	378
Looking at Variables	378
Compiling a Project	379
Testing a Shader	380
Benchmarking	380
Window Layouts	380
Using Your Shaders in an OpenGL Program	380
Creating a Shader	381
Creating a Shader Object	381
Loading a Shader	382
Reading Shader Source	382
Compiling the Shader	382
Creating a Program Object	383
Cleanup	383
OpenGL ES Performance Detective	384
Before You Run Performance Detective	384
Running Performance Detective	384
Viewing the Results	385

Acknowledgments

This is the part of the book that is more interesting to people who know me than for people purchasing this book. I acknowledge the people who helped me complete this book.

First I would like to thank the people who answered my questions, both those who answered my questions I posted on Apple's developer forums and those who indirectly answered my questions by answering questions other people posted on programming forums. I also extend my thanks to people who blogged about Xcode 4, Mac, and iOS programming. You allowed me to find the answers to my questions without me having to ask. I encourage anyone reading this book to share their knowledge through blogging or answering questions on programming forums. You help more people than you realize.

Second, I would like to thank the people who asked Xcode 4 questions on forums like Stack Overflow. Your questions helped me determine the material I needed to cover in the book.

Finally I would like to thank my family for their support: my parents Stan and Mary, my brothers Dave and Steve, my sister Kathy, my sister-in-law Rheia, my brother-in-law John, my nephews Zachary and Christian, and my nieces Alyssa and Danielle. I wouldn't have been able to write this book without you. I've been blessed to be part of such a loving family.

Introduction

The Xcode Tools contain everything you need to create Mac OS X and iOS applications. As powerful as the Xcode Tools are, they can be difficult to learn. Most Mac OS X and iOS development books focus on Cocoa and Cocoa Touch programming, as they should. They teach enough Xcode for you to create the projects in the book. But there's more to writing real applications than writing code. Real applications must be tested and debugged to make sure they run properly and must be profiled to make sure they run fast enough. A Cocoa or iOS programming book isn't going to show you how to profile your program or find memory leaks in it.

Xcode Tools Sensei picks up where other books leave off. It teaches the Xcode Tools, not a particular language or programming framework. By reading this book you can spend more time writing, debugging, and profiling your applications and less time searching and reading documentation.

Getting Xcode 4

Xcode Tools Sensei covers Xcode 4, which requires an Intel Mac running Mac OS X 10.6 or later. You can download the latest version of Xcode from the Mac App Store. When you download Xcode from the App Store, it is installed in your Applications folder.

Use Apple's developer site to download older versions of Xcode. The most common reason to download an older version of Xcode is that you're running an older version of Mac OS X. If you're running Mac OS X 10.7 or later, you must sign up for a free ADC membership to download Xcode from Apple's developer site. If you are running Mac OS X 10.6, you must be a member of one of Apple's paid developer programs, iOS or Mac, to download Xcode 4. Xcode 4.2 is the latest version that runs on Mac OS X 10.6.

Xcode 4.3 and later versions are packaged as a single application that you install in your Applications folder. Older versions require you to run an installer. In most cases after running the installer, you can find Xcode in the following location:

`/Developer/Applications`

Accessing Other Developer Tools

Starting with Xcode 4.3, Apple packages Xcode as a single application instead of creating a **Developer** folder on your startup disk. The change in packaging makes finding the other developer tools more difficult. You can launch other developer tools, such as Instruments, by choosing Xcode > Open Developer Tool.

The Mac OpenGL tools I cover in the book are not packaged with Xcode. Choose Xcode > Open Developer Tool > More Developer Tools to go to Apple's developer downloads page. Download the Graphics Tools for Xcode package to install the OpenGL tools. If you want to compile code from the command line, download the Command Line Tools for Xcode package.

The Book's Contents

Chapter 1 covers Xcode projects. In this chapter you will learn how to create a project, add files to a project, create workspaces, and use the Organizer. You will also learn about the types of projects you can create in Xcode and Xcode's project window.

Chapter 2 covers code editing. Some of the topics covered in this chapter include Xcode's editor, code completion, code snippets, tabbed editing, refactoring tools, and reading developer documentation.

Chapters 3 and 4 cover Interface Builder, which is now integrated with Xcode in Xcode 4. In Chapter 3 you will learn how to use Interface Builder to create user interfaces for Mac applications. In Chapter 4 you will learn how to use Interface Builder to create user interfaces for iOS applications.

Chapter 5 covers Xcode's data modeling tools, which work with the Core Data framework. In this chapter you will learn how to create data models and mapping models, which migrate data from an older version of a data model to a newer version.

Chapter 6 covers building projects. Some of the topics covered in this chapter include the project editor, targets, build settings, schemes, configuration settings files, and static analysis.

Chapter 7 covers debugging. In this chapter you will learn how to set up your scheme for debugging, set breakpoints, view variables, step through code, and use the debug console with both GDB and LLDB.

Chapter 8 covers version control using git and Subversion. Some of the topics covered in this chapter include creating a version control repository, adding an Xcode project to a repository, viewing changes in versions of a file, adding files to a repository, committing changes to a repository, branching, pushing to and pulling from remote git repositories, and snapshots.

Chapter 9 covers Instruments, which is a tool for tracing applications. With Instruments you can do things like check for memory leaks, find out how much memory your application is using, and determine where your application is spending its time.

Chapter 10 covers command-line debugging tools: `fs_usage`, `sc_usage`, `vmmmap`, `heap`, `leaks`, and `malloc_history`. If you've never heard of these tools before, don't worry. After reading Chapter 10 you'll become intimately familiar with them.

Chapter 11 covers the OpenGL tools: OpenGL Profiler, OpenGL Driver Monitor, OpenGL Shader Builder, and OpenGL ES Performance Detective. OpenGL Profiler profiles and debugs OpenGL applications. OpenGL Driver Monitor displays realtime statistics about your graphics card. OpenGL Shader Builder is the tool to create OpenGL shaders to give your OpenGL applications more control over drawing a scene. OpenGL ES Performance Detective investigates your OpenGL ES application for possible performance problems. If it finds a problem, OpenGL ES Performance Detective lists the most likely causes of the problem.

Xcode Preferences

Throughout the book I make references to Xcode's preferences, such as its Text Editing preferences. Choose Xcode > Preferences to open Xcode's preferences window. Xcode's preferences window has the following sections:

- General
- Behaviors
- Fonts and Colors
- Text Editing
- Key Bindings
- Downloads
- Locations

Older versions of Xcode may have Documentation and Source Trees preferences instead of Downloads preferences. Older versions of Xcode may also have Distributed Builds preferences.

What the Reader Needs to Know

I assume the reader has some experience writing Mac or iOS programs. Mac OS X and iOS development are large topics. There is no way I could adequately cover programming and the Xcode Tools in one book.

Those of you new to programming, Mac development, or iOS development should buy another book or two in addition to this book. You can find programming books and reviews at Amazon. I also provide links to publishers in the following post on my blog:

<http://meandmark.com/blog/2010/01/getting-started-with-mac-programming/>

Some Things to Keep in Mind as You Read This Book

The Xcode Tools have multiple methods to perform many tasks. Rather than detail each possible way to accomplish a task, I usually mention only one of the methods in the chapter text. Just because I mention one way to do something doesn't mean the other methods are worse. You may find it easier to complete a task differently than the way I explain in the book.

Apple is constantly making changes to the Xcode Tools. The constant changes mean that screenshots, labels, control names, and menu item names in the book may be different than what you see on your Mac.

I mention right-clicking many times in the book. Every Intel Mac should have either a mouse or a trackpad that allows right-clicking to open contextual menus. If your mouse or trackpad doesn't support right-clicking, hold down the Control key while clicking to open contextual menus.

Xcode 4.4 and 4.5 are available for both Mac OS X 10.7 and 10.8. Some XCode 4.4 and 4.5 features may not be available in Mac OS X 10.7. Xcode 4.1 and 4.2 are available for both Mac OS X 10.6 and 10.7. Some Xcode 4.1 and 4.2 features may not be available in Mac OS X 10.6.

For the latest updates go to this book's official website.

<http://www.meandmark.com/xcodebook.html>

If you have any questions or comments about the book, feel free to email me at the address below. I'll do my best to answer any questions.

xcodebook@meandmark.com

Chapter 1

Xcode Projects

Every program you create in Xcode requires a project, even a simple command-line program with one file. Because every program requires a project, covering projects is a good way to start the book. After reading this chapter you will know how to create a project and add files to a project. You will also become familiar with Xcode's project window.

Creating a Project

For most of you, the first you'll do when you start with Xcode is create a project. An Xcode project contains your program's source code files and other files, such as xib files, Xcode needs to build a working program. When you launch Xcode, a welcome window opens that gives you the option of creating a new project. You can also choose File > New > Project to create a new project. Creating a project is a three-step process.

Step 1: Choose the Type of Project You Want to Create

When you create a new project, the New Project Assistant opens, which you can see in Figure 1.1. On the left side of the window is a list of project categories. Xcode has the following project categories:

- Application
- Framework and Library
- Application Plug-in
- System Plug-in
- Other
- iOS Application
- iOS Framework and Library
- iOS Other, which lets you create an empty project or in-app purchase content.



I will go into greater detail on the types of projects shortly, but most of you will be making application projects. Selecting a category shows the projects you can create for a given category. When you select a project, a description appears at the bottom of the New Project Assistant. After choosing the type of project you want to make, click the Next button to move on to Step 2.

Step 2: Choose a Product Name

What you can do in Step 2 depends on the type of project you select in Step 1, but choosing a product name is one thing you will do for every project. The product name is the name of what Xcode builds. For an application the product name is the application name. The product name is also the name Xcode gives the project file.

Xcode 4.4 adds an Organization Name text field for most project templates. The name you enter in the text field appears in the copyright notice at the top of new source code files Xcode adds to the project.

Many Xcode project templates let you specify a company identifier along with the product name. The company identifier uniquely identifies you as the creator of the product. It takes the following form:

`com.CompanyName`

You can use your name as a company name if you don't work for a company.

If your project uses the Cocoa or Cocoa Touch frameworks, you will see a checkbox named Include Unit Tests. Selecting this checkbox tells Xcode to add a unit testing bundle target to your project. Select the checkbox if you're going to unit test your project.

Projects that use the Cocoa and Cocoa Touch frameworks should see a Use Automatic Reference Counting checkbox. Selecting the checkbox tells Xcode to use automatic reference counting, which simplifies memory management in Objective-C applications. I recommend using automatic reference counting for new projects.

Cocoa and iOS applications may see a Class Prefix text field. If you enter a prefix in this field, Xcode adds the prefix to any classes and class files you create in the project. An example of a class prefix is the NS prefix used in many Cocoa classes. If you enter XYZ for a Cocoa application project, Xcode adds the files `XYZAppDelegate.h` and `XYZAppDelegate.m` to the project. The name of the class in this case is `XYZAppDelegate`.

As I said at the beginning of this section, the choices you can make in Step 2 depend on the type of project you create. Some examples of the choices you can make include the following:

- Using Core Data in Cocoa applications and some iOS applications.
- The language to use for a command-line tool.
- The type of library to create for a library project: static or dynamic.
- Creating a document-based Cocoa application.
- Picking the device for iOS applications: iPhone or iPad.

After making your choices, click the Next button to move on to the final step.

Step 3: Save the Project

Tell Xcode where you want to store the project and click the Create button. Congratulations! You've created an Xcode project.

At the bottom of the Save panel you will see a checkbox named Create local git repository for this project. If you select this checkbox, Xcode creates a git repository for your project and places the project under version control. A version control system tracks the changes made to files and who made the changes. Read Chapter 8, "Version Control", for more information on version control.

If you're learning programming, you don't need to create a git repository for your project. Larger projects and projects with a long lifespan are the types of projects that would benefit from creating a git repository.

The Project's Contents

After creating a project, you should see its contents on the left side of the project window. What Xcode includes in a project depends on the type of project you create. Xcode includes the following files for a Cocoa application:

- Three Objective-C source code files: `main.m`, `AppDelegate.h`, and `AppDelegate.m`. The app delegate files may have the name of your project as a prefix.
- The Cocoa framework.
- A xib file, `MainMenu.xib`, containing the user interface. Select the xib file to build the user interface.
- Two property list files, `Info.plist` and `InfoPlist.strings`. The property list files may have the name of your project as a prefix, like `ProjectName-Info.plist`. Property list files are XML files that consist of key/value pairs. The key stores the property name, and the value stores the property's value. The `Info.plist` file contains information about the application, such as its name and version number. The `InfoPlist.strings` file contains localized application information, such as a copyright notice. There will be one `InfoPlist.strings` file for each spoken language your application supports.

- A prefix header, `ProjectName-Prefix.pch`. Xcode precompiles the header files in the prefix file, which makes your project compile faster.
- A rich text file, `Credits.rtf`, that lets you credit everyone who helped create the application.

Each project type includes its own set of files. A Core Data application is going to use the Core Data framework along with the Cocoa framework. It also will include a data model file with the extension `.xcdatamodeld`. A C++ command-line application won't include any frameworks or xib files. What's important to know is each of Xcode's project types provides a starting point for writing your own programs. You can focus on writing code rather than worrying about forgetting a framework.

Assuming you choose a Cocoa application project, you can run it without writing any additional code. If you're getting antsy to do something on the computer, you can test my previous statement by clicking the Run button on the left side of the project window toolbar. Xcode will compile the source code and run the application, which displays a blank window on the screen.

Application Projects

Application projects enable you to create Mac applications that can be launched from the Finder. You can create GUI applications using the Cocoa framework or command-line applications that run in the Terminal application. If you're learning programming or learning Mac development with Xcode, you'll be creating application projects.

Cocoa Projects

You have a choice of two Cocoa projects: Cocoa Application and Cocoa AppleScript Application. The Cocoa application project uses Objective-C as its programming language. Objective-C is Cocoa's native language and is the language to use if you're not sure about what language to use.

As its name implies, the Cocoa AppleScript application project lets you write a Cocoa application in AppleScript. AppleScript is good for small applications, but it's not suited for large applications. AppleScript was designed for writing scripts to automate other applications, not for writing large applications.

Document and Non-Document Applications

Cocoa applications give you the option of creating an application or a document-based application. The difference between the two application types is document-based applications can have multiple documents open at one time. Text editors, spreadsheets, and web browsers are examples of document-based applications; you can have five documents open at once. iPhoto is an example of an application; you don't open five windows with each one displaying a different photo. If your application creates a new window when the user chooses File > New, you should create a document-based application.

If you create a document-based Cocoa application, Xcode asks you for the name of the document class, which is a subclass of `NSDocument`. Xcode creates a source code file and a xib file with the document class name you supply. Xcode also asks you for a file extension for the documents your application creates.

Core Data Applications

Core Data application projects are Cocoa applications that use the Core Data framework. The Core Data framework makes creating data structures for your Cocoa applications easier. Core Data lets you use Xcode's modeling tools to define your program's data structures instead of writing code. Read Chapter 5, "Modeling Tools", for more information on Xcode's modeling tools.

If you decide to use Core Data in a Cocoa application project, you have the option to include a Spotlight importer. Your application would require a Spotlight importer if it uses a custom document format and you want Spotlight to be able to find your application's files when the user searches in the Finder.

Command-Line Tool Projects

Command-line tool projects create programs that run from the Terminal application instead of the Finder. If you're learning C, C++, or Objective-C, command-line tool projects are perfect. They let you use the standard C and C++ functions to print output on the screen. Those functions don't work with graphical user interfaces like Mac OS X's. By using a command-line tool project, you can learn the language without having to deal with the complexity of writing Mac programs. You can write command-line tool programs in C, C++, or Objective-C. Use the Type pop-up menu to choose the language. Choose Foundation to create an Objective-C program.

You can also create command-line tool projects that use the Core Data, Core Services, and Core Foundation frameworks. Most Core Data applications work with the Cocoa framework, but you can use Core Data to write a command-line program.

Core Services contains the operating system services that have nothing to do with a program's user interface, such as networking, file management, memory management, and multiprocessing. Higher-level frameworks like Cocoa sit on top of Core Services. The Core Foundation framework is the C language equivalent of the Foundation framework. It allows C and C++ programs to take advantage of the Foundation framework.

Framework and Library Projects

The projects in the Framework and Library group are meant for reusable code. The code could be used by you in multiple projects or it could be used by developers all over the world in their projects. There are three categories of projects in the Framework and Library group: libraries, frameworks, and bundles.

Libraries

You can write libraries in C, C++, or Objective-C. If your library is going to be used on operating systems other than Mac OS X, use a C or C++ library project.

When you create a library project, you must choose between creating a static or a dynamic library. A *static library* is linked to an application when compiling the application. A *dynamic library* is linked when the application runs. Static libraries are easier to use, but dynamic libraries can save memory if multiple running applications use the same library. Suppose you have a library that five running applications use. A static library would load five copies of the library, one for each application. A dynamic library would load one copy, shared by the five running applications.

Frameworks

Frameworks are similar to dynamic libraries, but they also contain header files and resource files. Frameworks are easier to use than dynamic libraries. The problem with frameworks is they work only on Mac OS X. Frameworks are a good choice for reusing Cocoa code.

Bundles and XPC Services

You use bundle projects most often to create plug-ins, programs other applications use to add features to the application. Lots of applications support plug-ins, such as iPhoto, Photoshop, Maya, and REAL Studio. Bundles can be created using the Cocoa and Core Foundation frameworks.

An XPC service provides a low-level method of interprocess communication that is compatible with Apple's Grand Central Dispatch technology. Apple's *Daemons and Services Programming Guide*, which is part of Apple's developer documentation, has more information on XPC services.

Application Plug-in Projects

The Application Plug-in group has project templates to create plug-ins for Automator, Address Book, Installer, and Quartz Composer. The Automator plug-in project lets you create Automator actions. An action is a loadable bundle that performs one task. Create workflows in Automator by linking actions together. Use workflows to automate tasks in your application and other applications. You can write Automator actions using AppleScript, Objective-C, or shell scripts.

System Plug-in Projects

The system plug-in projects are lower level than the application plug-in projects.

- Generic C++ plug-in
- Generic kernel extension
- Image unit plug-in
- IOKit driver
- Preference pane
- Quick Look plug-in
- Screen saver
- Spotlight plug-in
- Sync schema (removed in Xcode 4.4)

Use the generic C++ plug-in project if you need to write a plug-in that isn't covered by the other projects in the System Plug-in group.

The generic kernel extension project lets you write a kernel extension. Kernel extension programming is low-level operating system programming with the potential to wreck your computer so be careful if you choose to write a kernel extension.

The image unit plug-in project creates a plug-in for Core Image. Core Image is a Mac OS X technology for applying effects and filters to images.

Use the IOKit driver project to write drivers for peripherals like joysticks, gamepads, printers, and scanners.

The preference pane project lets you build a preference pane, which is a plug-in for the System Preferences application. Choose Apple > System Preferences to launch System Preferences.

If you have a custom document type, a Quick Look plug-in for that document type lets the user examine a document by choosing File > Quick Look in the Finder.

The screen saver project lets you write a Mac screen saver.

The Spotlight plug-in project is used to import metadata from a file. If your application uses a custom file type and you want it to be searchable in the Finder using Spotlight, create a Spotlight plug-in.

The sync schema project lets you build a schema to synchronize data between your application, other applications, and other devices. Suppose you have an application the user has installed on a desktop Mac. With a sync schema the user could synchronize their data with a laptop, cell phone, or iPod.

Other Projects

There are three possible templates in the Other group: empty project, external build system project, and in-app purchase content project. External build system projects use a program other than Xcode to build the projects. There are two cases where you would use an external build system project. The first case is when you want to use a different build system than Xcode's to build your project. This situation occurs most often when you're writing software for multiple operating systems. Build systems like `make`, `SCons`, and `CMake` let you build your program in a cross-platform manner. You can use one of these build systems to build a Mac application in Xcode, a Windows application in Visual Studio, and a Linux application in Eclipse.

The second case to use an external build system project is when you're writing a program in a language other than the ones for which Xcode supplies compilers: AppleScript, C, C++, and Objective-C. An external build system project lets you use Xcode to write code in any programming language, as long as you install that language's compiler on your Mac. Mac OS X ships with Java, Perl, Python, and Ruby so you don't have to install anything to use those languages.

When you choose to create an external build system project, Xcode asks you for the path to the build tool you're going to use to build the project. The initial tool is `make`. If you're going to use a different tool to build your project, enter the path to the tool in the Build Tool text field.

Xcode 4.4 adds the in-app purchase content project template for both Mac and iOS applications. Because in-app content requires an app, most of you will add an in-app purchase content target to an application project instead of creating a new project. Refer to the “Adding Targets” section in Chapter 6, “Building Projects”, for more information on adding targets to a project.

iOS Application Projects

The iOS Application group has project templates to write applications that run on the iPhone, iPod Touch, and iPad. Xcode has the following iOS application templates:

- Master-detail application
- OpenGL game
- Page-based application
- Single view application
- Tabbed application
- Utility application
- Empty application

Master-detail applications usually contain a master list of items. Selecting an item from the list displays information about the selected item. The master-detail application template includes a navigation controller to display the master list of items. The template also includes a split view for iPad applications.

The OpenGL game template includes the OpenGL ES framework. If you’re writing a game or an application that requires high-performance graphics, use the OpenGL game template.

The page-based application template includes a page view controller, which simplifies dealing with multiple pages of information in your application. The single view application template is for iOS applications that use only one view.

The tabbed application template includes a tab bar and a view controller for the tab bar. If your application uses a tab bar, choose the tabbed application template.

The utility application template is used to write simple applications that do not require much input from the user. An example of a utility application is an application that provides a five-day weather forecast.

The empty application template provides a window and an application delegate. You can use this template as the starting point for any iOS application.

Devices

When you choose a product name for your iOS application project, you should see a Devices pop-up menu. The menu determines the devices your application supports and determines the xib file Xcode creates for you. If you pick iPad, Xcode gives you an iPad xib that includes an iPad-sized main window. If you pick iPhone, Xcode gives you an iPhone xib that includes an iPhone-sized main window. If you pick Universal, Xcode creates two xib files: one for iPhone and one for iPad.

Creating a universal application is easy. All of the iOS application templates can create a universal application that runs on both iPhones and iPads. Choose Universal from the Devices menu.

Core Data

Several of the iOS application templates give you the option to use Core Data. Core Data lets you use Xcode's modeling tools to define your program's data structures instead of writing code. If you can use Core Data, there will be a Use Core Data checkbox below the Product Name text field. Select the checkbox to use Core Data. If you decide to use Core Data, Xcode adds the Core Data framework and a data model to the project.

Storyboarding

When naming your project, you should see a Use Storyboards checkbox. Selecting the checkbox tells Xcode to add storyboard files to your project instead of xib files. Storyboards allow you to view all your application's screens in one place. Read the section "Storyboarding" in Chapter 4, "Creating User Interfaces for iOS Applications", for more information on storyboarding.

iOS Library Projects

The library project template lets you create a static library using the Foundation framework.

Project Window

The project window, shown in Figure 1.2, is your project's home when working in Xcode. It has the following components:

- Toolbar
- Navigator
- Editor
- Utility area
- Debug area

All five areas of the project window may not be visible initially. Use the View menu or the View buttons on the project window toolbar to control what appears in the project window. The View menu also lets you go to specific areas of the project window, such as going to a specific navigator.

Toolbar

At the top of the project window is the toolbar, which provides you easy access to tasks you perform most. On the left side of the toolbar are controls related to running your program in Xcode. There are buttons to run your program and stop running it. Next to the Run and Stop buttons is the Scheme menu. A scheme lets you control how Xcode builds, tests, and launches your project. I cover schemes in more detail in Chapter 6, “Building Projects”; read the “Schemes” section.

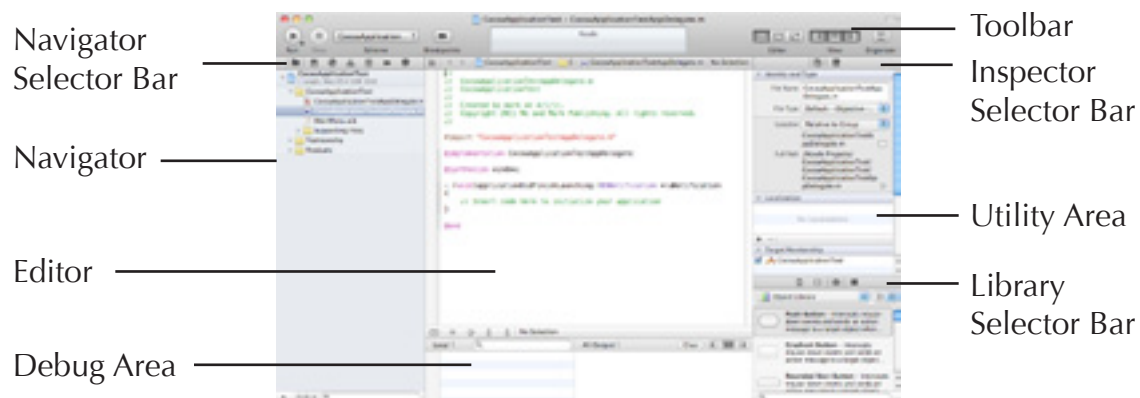


Figure 1.2

Project window

Next to the Scheme menu is a Breakpoints button that toggles your breakpoints. Initially any breakpoints you set are deactivated. Clicking the Breakpoints button activates the breakpoints you set, and Xcode runs your project in the debugger when you click the Run button. Clicking the button a second time deactivates the breakpoints, and Xcode runs your project outside the debugger. Chapter 7, “Debugging”, has more information on debugging and breakpoints.

In the center of the toolbar is the status area. It shows the progress of time-consuming tasks like downloading documentation.

On the right side of the toolbar are three sets of buttons: Editor, View, and Organizer. The Editor group has three buttons and tells Xcode what editor to use: standard, assistant, or version. The standard editor gives you one large area for editing. The assistant editor splits the editor in two. For source code files the assistant editor shows the file and its counterpart. If you select a header file in the navigator, the assistant editor shows both the header file and its corresponding implementation file. The version editor works only with files under version control. It lets you compare versions of a file. You can also use the View menu to pick your editor by choosing View > Standard/Assistant/Version Editor > Show Standard/Assistant/Version Editor.

The View group of buttons determines what project views are visible. The first button toggles the navigator. The second button toggles the debug area. The third button toggles the utility area. You can also use the View menu to hide and show the navigator, debug area, and utility area.

The Organizer group isn’t really a group because it has only one button. Clicking it opens the Organizer window. Read the “Organizer” section later in this chapter for more information on the Organizer.

Navigator

Xcode projects have a lot of information to track. The information you want to access depends on what you’re currently doing. When you’re writing code, you want access to the files in the project. When you build your project, you want access to compiler errors and build logs. When debugging, you want access to the breakpoints you set. The navigator lets you quickly access different areas of your project.

Above the navigator is the navigator selector bar. The navigator selector bar has buttons to pick the navigator to show. The selector bar has the following buttons, running from left to right:

- Project navigator
- Symbol navigator
- Search navigator
- Issue navigator
- Debug navigator
- Breakpoint navigator
- Log navigator

You can also go to a specific navigator by choosing View > Navigators.

Project Navigator

When you create a new project, the project navigator is the navigator you see. The project navigator shows the files in your project. Selecting a file from the project navigator opens the file in the editor.

The project navigator initially shows all the files in the project. At the bottom of the navigator are controls to filter what appears in the project navigator. The search field lets you enter text. If you enter `.m` in the search field, the project navigator shows all the Objective-C implementation files in your project.

Next to the search field are three tiny buttons. Clicking the first button tells Xcode to show recently edited files in the project navigator. Clicking the second button tells Xcode to show files with source control status, files that are different from the version in the version control repository. If your project is not under version control, clicking the second button hides all the files in the project navigator. Clicking the third button tells Xcode to show only files with unsaved changes.

Project Navigator Groups

When you examine your project's contents in the project navigator, you should see some folders. These folders are groups, which help you organize your project's files in Xcode. Groups do not correspond to folders in the file system; they exist only in Xcode.

You can create your own groups by right-clicking in the project navigator and choosing New Group. If the New Group menu item is disabled, make sure you right-click on a file or folder. The menu item is disabled if you right-click below the project's files. Selecting some files, right-clicking, and choosing New Group from Selection creates a new group and places the selected files in the group. Dragging a file from the project navigator to the folder adds the file to the group.

Select the group in the project navigator and either click or press the Return key to rename the group. Select the group and press the Delete key to remove the group. If you have files inside the group, deleting the group also deletes the files in the group. If you don't want to delete the files, drag them out of the group before you delete the group.

Symbol Navigator

The symbol navigator allows you to view the symbols in your projects. Examples of symbols are classes, variables, functions, data structures, and enumerated data types. The symbol navigator has the following groups, shown in Table 1.1:

Xcode initially shows only classes in the symbol navigator. To see the other groups, go to the bottom of the symbol navigator. There is a group of three tiny buttons next to the search field. Click the left button, the one with the letter C, to show the other groups.

For each group listed in Table 1.1, Xcode tells you the number of that group defined in the project and the number defined in the system. Examples of system-defined symbols are symbols from the Cocoa framework and symbols from the Standard C Library. Normally the number of system-defined symbols is larger than the number of project-defined symbols. A small Cocoa application might have 5-10 classes in the project and have several hundred system classes because the Cocoa framework is large.

Table 1.1 Symbol Navigator Groups

Symbol Type	Description
Classes	Classes and their members. The Classes group is the one Objective-C and C++ developers will use the most.
Protocols	Objective-C protocols. A protocol is a list of methods for another class to implement.
Functions	Functions that are not member functions of a class, such as C functions.
Structs	C structures.
Unions	C unions, which are similar to structs, but only one field of the data structure is stored in memory at one time.
Enums	Enumerated data types that use the <code>enum</code> statement.
Types	User-defined data types. If you use the <code>typedef</code> statement to define data types, they appear under Types.
Globals	Global variables.

Click the disclosure triangle next to an entry to see its contents. Xcode initially shows only project-defined symbols. Selecting a symbol from the symbol navigator opens the symbol's file in the editor and takes you to the symbol.

There are two buttons above the symbol navigator: Hierarchical and Flat. Clicking the Hierarchical button shows a hierarchical symbol listing. Clicking the Flat button provides one flat list of symbols in alphabetical order.

Filtering the Symbol Navigator's Contents

There are two ways to filter what appears in the symbol navigator. First, you can enter text in the search field. Symbols matching the text you entered appear in the symbol navigator.

Second, you can use the three tiny buttons next to the search field to filter. The buttons from left to right are the following:

- Show only class symbols. Clicking this button shows a list of classes and hides the other groups. Showing only class symbols can help for Objective-C applications that don't have many entries for anything besides classes.
- Show only project-defined symbols. Clicking this button hides system-defined symbols. Showing only project-defined symbols allows you to focus on the symbols in your code. It also improves symbol navigator performance because Xcode doesn't have to show the system-defined symbols.
- Show containers only. Clicking this button affects the Classes group. Showing containers lists each class in the symbol navigator, but does not show the members of each class.

Search Navigator

The search navigator is used for project-wide search. Choose Edit > Find to search a single file. Start entering your search term in the search field. A menu opens that has two sections: search in project and search in project and frameworks. In both sections you can find text containing the search term or find text starting with the search term. Finish entering the search term and either choose an option from the menu or press the Return key.

Xcode groups the search results by file. Selecting a search result opens the file in the editor and takes you to the result.

Customizing Your Search

Clicking the magnifying glass button in the search field and choosing Show Find Options brings up options in the search navigator for you to customize your search. You have the following options:

- Style: textual or regular expression. A regular expression is a string that describes a search pattern. If you are unfamiliar with regular expressions, use a textual search.
- Hits must: contain, start with, match, or end with search term.
- Case: ignore or match. Suppose you enter the search term `string`. If you ignore the case the search navigator displays results for `string`, `String`, and `STRING`. If you match case, the search navigator displays results for `string` only.
- Find in: workspace or custom. The “Find Scopes” section shows how to create a custom find.
- Checkbox to search linked frameworks. Searching linked frameworks can bring up lots of results.

Xcode 4.4 adds two styles for searching your project’s symbols: symbol definitions and symbol references. When you use symbol definitions, the search navigator generates a listing for each place in your code that defines the symbol. When you use symbol references, the search navigator generates a listing for each place in your code that refers to the symbol. Symbol references generate more listings than symbol definitions.

Suppose you search for a function in your project named `DoSomething`. If you search symbol definitions, the search navigator displays one listing where you defined `DoSomething` in the header file and another listing where you wrote the code for `DoSomething` in the implementation file. If you search symbol references, the search navigator creates the two listings it creates for symbol definitions, plus a listing for each place in your code that calls `DoSomething`.

Xcode 4.4 adds the ability to insert a pattern into the search field. Click on the magnifying glass icon in the search field and choose Insert Pattern. Inserting a pattern allows you to search for things like tabs, white space, hexadecimal digits, email addresses, and URLs. You can add patterns to a search term to fine tune the search results.

Find Scopes

Create a find scope when searching in the workspace is not good enough. To create a find scope, choose Custom from the Find in menu in the search navigator.

When you choose Custom from the Find in menu, a sheet opens where you create find scopes. Figure 1.3 shows an example of the find scope sheet. On the left side is a list of find scopes. The list is empty if this is the first find scope you're creating. Click the + button to add a find scope. Double-click a scope entry to change its name. The scopes you create end up in the Find in menu.

Selecting a find scope from the list displays a list of conditions that must be met to show up in the search results. Click the + button to add a condition. The find scope editor has the following conditions for you to add:

- Location, which lets you search files based on location: within workspace or within a file or folder. Searching within the workspace only makes sense if you're combining other conditions because you can search within the workspace without creating a find scope. Searching within a folder would help if you wanted to search in a subfolder of your project. If you choose to search within a file or folder, click the Choose Path pop-up cell to pick a file or folder.
- Name, which lets you search files based on file name. Searching by name has the following conditions: is equal to, is not equal to, contains, starts with, ends with, or matches regex. Regex is a regular expression. Enter the term in the text field.
- Path, which lets you search multiple paths. A path has the same conditions as Name.
- Path Extension, which lets you search files based on file extension. A path extension has the same conditions as Name and Path.
- Type, which lets you search files based on file type. The Type condition has two pop-up cells. The first cell lets you specify is or is not. The second cell lets you specify the type of file: source code, script, a specific language, HTML, XML, any, or other type. Choose Other Type to specify a file type not listed in the pop-up cell. Enter the file extension in the text field.

Find and Replace

To do find and replace, click the Find menu next to the search field and choose Replace. An unlabeled text field appears below the search field along with Preview, Replace, and Replace All buttons. Enter the replace term in the text field. The navigator shows the search results.

Clicking the Preview button lets you preview the changes, which you can see in Figure 1.4. In the preview window there is a listing for each search match in the project with a checkbox next to it. Selecting the checkbox tells Xcode to change that instance. Next to the listings are two panes. The left pane shows what the code will look like if you replace the text. The right pane shows the current code. There is a switch between the two code fragments with an indicator. The indicator pointing left tells Xcode to change that instance. Click the Replace button to make the changes.

Issue Navigator

The issue navigator lists any compiler errors, warnings, and static analyzer issues in your project. The sections “Seeing More Build Details” and “Static Analysis” in Chapter 6, “Building Projects”, have more information on the issue navigator.

Debug Navigator

The debug navigator lists the call stack when debugging your program. Read the “Debug Navigator” section in Chapter 7, “Debugging”, for more information on the debug navigator.

Breakpoint Navigator

The breakpoint navigator lists the breakpoints you have set in your project. Read the section “Setting Breakpoints” in Chapter 7, “Debugging”, for more information on the breakpoint navigator.

Log Navigator

The log navigator lets you view the logs of tasks you perform on a project, such as building the project, running it, and committing files to a version control repository. Select a log from the list to open it in the editor.



Figure 1.3

Find scope sheet



Figure 1.4

Find and replace preview

Editor

What you do in the editor depends on the file you select in the project navigator. If you select a source code file, the editor lets you edit the source code. If you select a xib file, you can modify your project's user interface. If you select the project file, you can edit project and target settings.

As I explained in the “Toolbar” section, Xcode has three editors: standard, assistant, and version. The standard editor provides one editing area and is the right editor to use to edit non-source code files. The assistant editor provides multiple editing areas. It is most useful for editing source code files and making connections to your code in xib files. The version editor compares two versions of a file. Your project must be under version control to use the version editor.

I cover the assistant editor in more detail in Chapter 2, “Editing Source Code”, specifically in the “Assistant Editor” section. I cover the version editor in more detail in Chapter 8, “Version Control”, specifically in the “Seeing the Changes You Made to a File” section.

Utility Area

The utility area contains inspectors and libraries. Inspectors let you tweak settings for a file. Use the inspector selector bar (see Figure 1.2) to pick an inspector. The inspectors that are available depend on the file type, but every file has at least two inspectors: File and Quick Help.

File Inspector

The file inspector shows information about a file. Select a file from the project navigator to see its information in the file inspector. The file inspector has the following sections for text files: Identity and Type, Localization, Target Membership, Text Settings, and Source Control.

Identity and Type

The Identity and Type section lets you specify a file's name, type, and location. The Location pop-up menu determines how Xcode stores the file's location. There are six options.

- Relative to group, which means Xcode uses the file's group in the project navigator to store the file's location.
- Relative to project, which means Xcode uses the project's folder to store the file's location.
- Absolute path, which means Xcode uses the file's path on your computer to store the file's location.
- Relative to build products, which means Xcode uses the folder where it places build products, such as executable files and libraries, to store the file's location.
- Relative to Developer directory, which means Xcode uses the folder where you installed Xcode to store the file's location.
- Relative to SDK, which means Xcode uses the folder where you installed the current SDK to store the file's location.

The default location type for files is by group. The reference type options matter if you're going to share your projects with other people. If you refer to your files according to the absolute path on your computer, chances are high that other people using your project have a different path to the files. In this case Xcode would be unable to find the files and the project would not compile. In most cases you should choose either Relative to Group or Relative to Project from the Location pop-up menu.

Localization

The Localization section lets you add a localization for a file. This section will be blank for source code files. You shouldn't have to add a localization for source code files. Files you would need to localize have one localization for the language you're using, which is English for most of you.

Target Membership

The Target Membership section tells you the targets the file belongs to. If you need a file to be a member of the target, select the checkbox next to the target.

Source code files, xib files, and files that are copied to the application bundle are the files in your project that are members of targets. Header files are normally not target members.

Text Settings

The Text Settings section let you control the text encoding, line endings, indentation, and line wrapping in the editor. In most cases you should use Xcode's Text Editing preferences to control these settings. The Text Settings section of the file inspector lets you control the text settings for a single file or for a single project. Select the project file from the project navigator to make changes to the text settings for the project. The section "Text Editing Preferences" in Chapter 2, "Editing Source Code", provides more information on Xcode's Text Editing preferences.

Source Control

The Source Control section tells you the version of the file and its source control status. If the file's status is modified, there is a Discard button that lets you discard the changes you made to the file.

Your project must be under version control to have a Source Control section. The section "Seeing Which Files Have Changed in Your Project" in Chapter 8, "Version Control", provides more information on source control status.

Quick Help Inspector

If you select some text in the editor, the Quick Help inspector shows the Quick Help documentation for the selected text. For Cocoa and Cocoa Touch classes, the Quick Help documentation provides an overview of the class or method as well as links to documentation. The "Quick Help" section in Chapter 2, "Editing Source Code", contains more information on Quick Help.

Library

Below the inspectors is the library. The library has content you can add to your project. Use the library selector bar (see Figure 1.2) to pick a library. There are four libraries: file template, code snippet, object, and media.

The file template library contains templates for blank files you can add to your project. Select a file from the library and drag it to the project navigator to add the file to your project. Read the section "Creating New Files for the Project" later in this chapter for information on the files you can add to your project.

The code snippet library contains code snippets. Drag a snippet from the library to the editor to add the snippet to your code. Read the “Code Snippets” section in Chapter 2, “Editing Source Code”, for more information on the code snippet library.

The object library contains user interface elements. Use the object library to build your application’s user interface. Read the “Object Library” section in Chapter 3, “Creating User Interfaces for Mac Applications”, for additional information on Mac user interface elements. Read the “Object Library” section in Chapter 4, “Creating User Interfaces for iOS Applications”, for more information on iOS user interface elements.

The media library contains graphics, audio, and video files. Use the media library to play audio and video files and to view graphics files. To have entries in the media library, you must either add media files to your project or open a xib file in a Mac project. Read the “Media Library” section in Chapter 3, “Creating User Interfaces for Mac Applications”, for more information on the media library.

Debug Area

The debug area of the project window becomes important when debugging your program. It shows your program’s variables, the debug console, and controls for controlling the execution of your program. I cover the debug area in more detail in Chapter 7, “Debugging”.

Adding Files and Frameworks to Your Project

Xcode includes a source code file when you create a new project, but unless you’re writing a very simple program, you’re going to be adding files to your project. Examples of files you add to a project include source code files, xib files, data files, graphics files, and audio files. You can create new files and add existing files to your project.

Creating New Files for the Project

To create a new file and add it to your project, choose File > New > File or drag a file from the file template library to the project navigator. Creating a new file consists of two steps: choosing a file type and naming the file. Dragging a file from the file template library to the project navigator is the equivalent of choosing a file type.

Choosing a File Type

When you create a new file by choosing File > New > File, the New File Assistant opens, which you can see in Figure 1.5. The left side of the assistant contains a list of template categories. Selecting a category fills the sheet with the category's templates. Selecting a file type from the list displays a description of the file type.

Select the type of file you want to create from the list. Click the Next button to move on to naming the file. Some file types have an intermediate step before naming the file. When you add an Objective-C class, you specify the new class's superclass before naming the file.

Naming the File

After selecting the type of file you want to create, click the Next button, which will take you to the final part of the file creation process. Name your file. Notice that Xcode automatically includes the appropriate ending based on the type of file you create. A C file will have the extension `.c`.

If you create an Objective-C file, you don't explicitly name it. Suppose you add an Objective-C class to your project. The New File Assistant has a Class text field. The class name you enter is the name of the file.

After naming the file tell Xcode where you want the file to reside on your hard disk. By default it will be in the folder where your project is. You also have the options to determine which group in your project to place the file and determine the targets in your project where the file should be added.

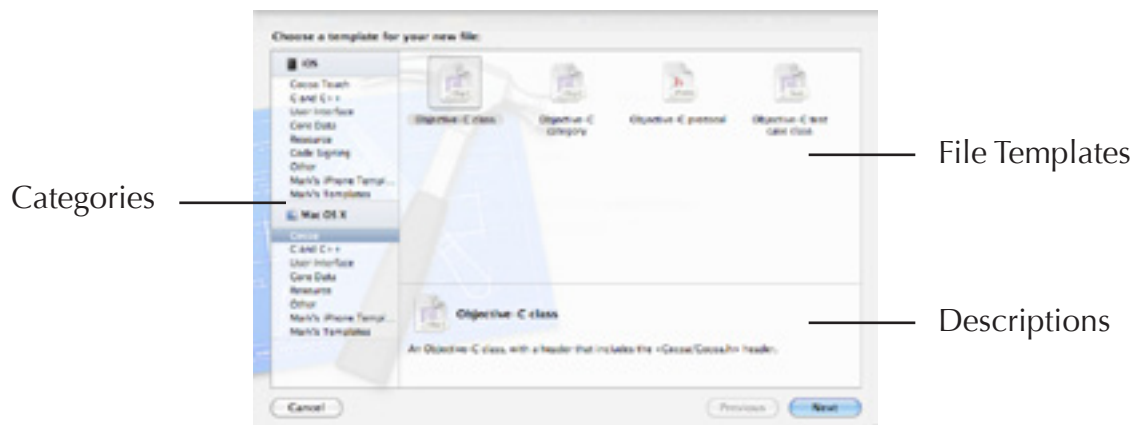


Figure 1.5

New File Assistant

Click the Save button to create the file. At the top of the file is a comment containing the following information:

- The file name.
- The project name.
- A notice saying who created the program and when.
- A copyright notice.

Mac File Types

Xcode 4 has the following file template groups for Mac OS X programs:

- Cocoa
- C and C++
- User Interface
- Core Data
- Resource
- Other

Cocoa

The Cocoa group contains Objective-C source code files. There are five types of Cocoa files you can add to your project: Objective-C class, Objective-C category, Objective-C protocol, Objective-C test case class, and Objective-C class extension. A category adds methods to an existing class, which lets you add methods to Cocoa classes and lets you break up large classes into multiple files. A protocol is a header file that contains a list of methods to implement. Any class can implement the protocol. A test case class is used with unit testing, which tests the methods in a class to make sure they're correct.

If you create an Objective-C class, you must specify the superclass, the class from which your newly created class inherits. The combo box to choose a superclass has the following superclasses: `NSObject`, `NSDocument`, `NSView`, `NSViewController`, and `NSWindowController`. If you want a different superclass, enter it in the combo box's text field. Inherit from `NSObject` if you're not sure of what superclass to use.

When you create an Objective-C category, you get to specify the class on which you're creating the category. Enter the class in the Category on text field.

If you create an Objective-C test case, make sure you have a unit test bundle target in your project. Choose File > New > Target to add a unit test bundle target to your project. Make sure the test case gets added to the unit test bundle target, not your other targets.

Apple added the Objective-C class extension file template in Xcode 4.4. A class extension file is a header file that allows you to declare part of a class's interface outside of the class's public header file. The interface is the `@interface` section in Objective-C header files. The main use of a class extension is to place private variables you don't want other developers to see. Place the variables you want other developers to see in the class's public header file, and place the ones you don't want them to see in the class extension file.

C and C++

There are three C and C++ files you can add to your project: C file, C++ file, and header file. If you add a C++ file, Xcode creates a corresponding header file for you.

User Interface

Use the User Interface group to add a xib file to your project. There are five xib files to choose from for Mac applications: application, empty, main menu, view, and window. When you create a Cocoa application project, the project contains a xib file with a main menu so you shouldn't need to create an application xib or a main menu xib. You're most likely to create a view xib file or a window xib file.

Core Data

The Core Data group contains files for Core Data applications. The data model and mapping model files allow you to use Xcode's data modeling tools in Core Data applications. Refer to Chapter 5, "Modeling Tools", for more information on data models and mapping models.

In addition to the data model and mapping model files, the Core Data group has a `NSManagedObject` subclass. Core Data entities must inherit from `NSManagedObject` or a `NSManagedObject` subclass. Creating a `NSManagedObject` subclass lets you create a custom class for your entities to inherit from.

Resource

The Resource group has three files: property list, RTF File, and strings file. A property list is an XML file that stores persistent hierarchical application data. They are useful for storing small amounts of strings and numbers. A common case where you would use a property list is to store user preferences in your application.

An RTF file is a text file that can contain styled text.

As you can tell from its name, a strings file contains a list of strings. Strings files allow you to keep string data separate from your code. They are useful if your application supports multiple spoken languages like English, French, Spanish, and Japanese. Place your text in strings files to make your localization go more smoothly.

Other

The files in the Other group don't fit into the other groups. If you need an assembly language file, an empty file, or a shell script for your project, you'll find them in the Other group.

An exports file contains a list of exported symbols, such as class, function, and variable names. Xcode normally exports all the symbols from your project. Unless other applications are going to be calling the functions in your project, you can stick with Xcode's default behavior and not bother with exports files.

If you want to limit the symbols your project exports, add an exports file to your project and add the symbols you want to export to the exports file. Use the name of the exports file as the value of the Exported Symbols File build setting, which is part of the Linking collection. Framework and library projects are the projects most likely to use an exports file.

A configuration settings file is a text file that contains build settings. If you find yourself changing some build settings for each project, place those settings in a configuration settings file. I have more information about configuration settings files in the "Configuration Settings Files" section in Chapter 6, "Building Projects".

iOS File Types

Xcode has the following groups for iOS projects:

- Cocoa Touch
- C and C++
- User Interface
- Core Data
- Resource
- Other

Cocoa Touch

The Cocoa Touch group contains Objective-C source code files. You can create an Objective-C class, an Objective-C category, an Objective-C protocol, an Objective-C test case class, and an Objective-C class extension. A category adds methods to an existing class, which lets

you add methods to Cocoa Touch classes and lets you break up large classes into multiple files. A protocol is a header file that contains a list of methods to implement. Any class can implement the protocol. A test case class is used with unit testing, which tests the methods in a class to make sure they're correct.

If you create an Objective-C class, you must specify the superclass, the class from which your newly created class inherits. The combo box to choose a superclass has the following superclasses: `NSObject`, `UIView`, `UIViewController`, `UITableViewCell`, and `UITableViewController`. If you want a different superclass, enter it in the combo box's text field. Inherit from `NSObject` if you're not sure of what superclass to use.

When you create a `UIViewController` or `UITableViewController` subclass, you have options to target the controller for iPad and add a xib file for the view controller's user interface.

When you create a `UIViewController` subclass, you have options to target the controller for iPad, create a `UITableViewController`, and add a xib file for the view controller's user interface. Use the Subclass of combo box to create a `UITableViewController`.

If you create an Objective-C category, you get to specify the class on which you're creating the category. Enter the class in the Category on text field.

If you create an Objective-C test case, make sure you create a unit test bundle target first. Choose File > New > Target to add a unit test bundle target to your project. Make sure the test case gets added to the unit test bundle target, not your other targets.

Apple added the Objective-C class extension file template in Xcode 4.4. A class extension file is a header file that allows you to declare part of a class's interface outside of the class's public header file. The interface is the `@interface` section in Objective-C header files. The main use of a class extension is to place private variables you don't want other developers to see. Place the variables you want other developers to see in the class's public header file, and place the ones you don't want them to see in the class extension file.

C and C++

There are three C and C++ files you can add to your project: C file, C++ file, and header file. If you choose to create a C++ file, Xcode creates a corresponding header file for you.

User Interface

Use the User Interface group to add a xib file or storyboard file to your project. There are four xib files to choose from for iOS applications: application, empty, view, and window. Since iOS application projects include a xib file, you most likely will not need to create an application xib. View and window xib files are the ones you're most likely to create.

Choosing the storyboard file adds a blank storyboard file to the project. Read the section "Storyboarding" in Chapter 4, "Creating User Interfaces for iOS Applications", for more information on storyboarding.

When you select a file from the User Interface group and click the Next button, you should see a Device Family menu. This menu determines whether Xcode adds an iPhone xib or an iPad xib to your project.

NOTE

The xib files for iOS in the file template library are iPhone-sized. To create an iPad-sized xib file, choose File > New > File.

Core Data

The Core Data group contains files for Core Data applications. The data model and mapping model files allow you to use Xcode's data modeling tools in Core Data applications. Refer to Chapter 5, "Modeling Tools", for more information on data models and mapping models.

In addition to the data model and mapping model files, the Core Data group has a `NSManagedObject` subclass. Core Data entities must inherit from `NSManagedObject` or a `NSManagedObject` subclass. Creating a `NSManagedObject` subclass lets you create a custom class for your entities to inherit from.

Resource

The Resource group has the following files: settings bundle, property list, RTF File, strings file, GPX file, and GeoJSON file. A settings bundle specifies the application's settings.

A property list is an XML file that stores persistent hierarchical application data. They are useful for storing small amounts of strings and numbers. A common case where you would use a property list is to store user preferences in your application.

An RTF file is a text file that can contain styled text.

As you can tell from its name, a strings file contains a list of strings. Strings files allow you to keep string data separate from your code. They are useful if your application supports multiple spoken languages like English, French, Spanish, and Japanese. Place your text in strings files to make your localization go more smoothly.

GPX files specify a route or location to simulate. Use a GPX file to simulate running an application from a different city than your current location.

Apple added the GeoJSON file template in Xcode 4.5. Use a GeoJSON file to store geographic data when generating routes from one location to another.

Other

The files in the Other group don't fit into the other groups. If you need an assembly language file, an empty file, or a shell script for your project, you'll find them in the Other group.

The Other group also contains a resource rules file. Resource rules files are optional. They contain instructions for copying resources like audio and graphics files to the application bundle when building the application. You would need a resource rules file if you wanted to override the default method of copying resources to the application bundle.

The iOS Other group contains one additional file: configuration settings file. A configuration settings file is a text file that contains build settings. If you find yourself changing some build settings for each project, place those settings in a configuration settings file. I have more information about configuration settings files in the "Configuration Settings Files" section in Chapter 6, "Building Projects".

Fixing the Copyright Notice

In Xcode 4.4 Apple added an Organization Name text field to the New Project Assistant that specifies the name to appear in the copyright notice of the files you create. Adding the Organization Name text field reduces the need to fix the copyright notice, but if you created your project in an older version of Xcode, the material in this section may help you.

If you look at the copyright notice in the files you create, it may contain the text MyCompanyName. Unless you happen to work for a company called MyCompanyName, you want Xcode to use either your name or your company name in the copyright notice. To change the company name, open System Preferences and click the Users & Groups button. Select your account and click the Address Book Card Open button. Add a company name to your card.

When you use Address Book to change the company name, every subsequent file you create uses that company name. But you may want the files in a single project to have a different company name. You might be hired by another company to write a program for them, and you want that company's copyright to appear in the program you're writing.

To set the company name for a single project, select your project file in the project navigator. Open the file inspector by choosing View > Utilities > Show File Inspector. In the Project Document section of the file inspector is the Organization text field. Use the Organization text field to change the company name for this project. Any source code files you create for the project will place the new company name in the copyright notice.

Adding Files You've Already Created

Many applications contain more than just source code. They contain graphics, sound, and data files. When working with graphics, sound, and data files, you normally create them in another application and add them to your Xcode project so they get bundled with your application when Xcode builds it. You also might have existing source code files you want to add to a project. To add files you've already created to a project, choose File > Add Files to ProjectName. An Open panel opens that lets you find the files you want to add. After selecting files to add, click the Add button to add the files.

At the bottom of the Open panel are options for controlling how the files are added, which you can see in Figure 1.6. Selecting the Copy items into group's destination folder (if needed) checkbox tells Xcode to copy the files you added into your project's folder if the files aren't already there.

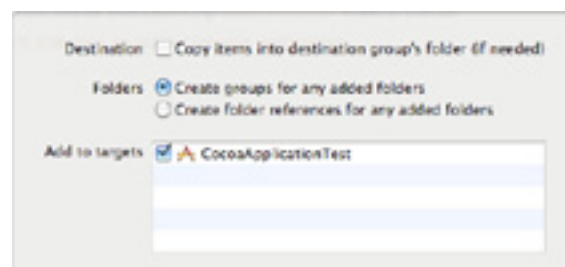


Figure 1.6

Options for adding existing files and folders to a project.

Adding a Folder of Files

If you're adding a folder of files, you have two options: recursively create groups for any added folders and create folder references for any added folders. When you recursively create groups, Xcode creates a group in the project navigator for each folder you add. Recursively create groups when you need to access the folder's files in Xcode. If you're adding a folder of source code files, you should recursively create the group because you want to be able to examine and edit the source code files.

When you create a folder reference, Xcode adds the folder to the project, but not the files in the folder. Create a folder reference when you don't need to access the folder's files in Xcode. If you're writing a game and you have a folder named Sound Effects that holds the game's sound effects, create a folder reference when adding the Sound Effects folder to your project. You're not going to be editing the sound files in Xcode. All you want to do is copy the folder to the application bundle when you build the project, and creating a folder reference lets you accomplish this task.

Adding Files to Targets

Below the adding folder options is a list of the targets in your project with a checkbox next to each target. If you select the checkbox for a target, the files you add will be added to that target. For an application target, files that you want in the application bundle should be added to the targets. Examples of files that should be added to the target are source code files, xib files, and image files. Header files, Xcode configuration settings files, and precompiled headers are examples of files that should not be added to the target.

Adding Frameworks and Libraries to a Project

If you're writing a program consisting mostly of GUI code, you can get away with using just the frameworks that are included when you create a Cocoa or Cocoa Touch application project. Many popular Apple technologies, such as OpenGL, GameKit, and Core Audio, require you to add a framework to your project to use the technology in your application.

Take the following steps to add a framework or library to your project:

1. Open the project editor by selecting the project file in the project navigator.
2. Select a target from the target list on the left side of the project editor.
3. Click the Build Phases button at the top of the editor.
4. Click the disclosure triangle next to the Link Binary with Libraries build phase.
5. Click the + button to add frameworks and libraries to the target.
6. A sheet opens with a list of frameworks and libraries. Select the frameworks you want to add, and click the Add button.

After adding frameworks to your project they will appear under the project in the project navigator. Drag the frameworks to the Frameworks folder if you want.

Source Trees

A source tree is a root path to place files that you want multiple people to work on. Source trees allow your project's files to remain independent of the project folder. The point of using source trees is to allow multiple people to work on a project. You can transfer the project to other people's computers without messing up the project's file references. To create a source tree, open Xcode's Locations preferences, click the Source Trees tab, and click the + button in the lower left corner of the preferences window.

There are three pieces of information you must supply to add a source tree.

- Setting Name is the name of the tree. Everyone who wants to use a source tree must give it the same setting name on their Macs.
- Display Name is the name Xcode shows for the source tree.
- Path is the location of the source tree on your hard disk.

Each person who wants to use a source tree can store it wherever they want on their hard disk. As long as everyone uses the same name for the source tree, everyone can use it. If you want other people to work on your projects, copy the files from the location of the source tree on your Mac to the location of the source tree on their Macs.

Removing Files from a Project

To remove a file from a project, select the file from the project navigator and press the Delete key. An alert opens asking if you want to delete the file. One of the buttons is Remove Reference Only. If you click this button Xcode removes the file from the project but does not delete it.

The second button depends on the version of Xcode you're running. Newer versions of Xcode have a Move To Trash button. Clicking this button removes the file from the project and moves it to the Trash.

Older versions of Xcode have a Delete button. Clicking the Delete button removes the file permanently from your hard disk. It does not move the file to the Trash. If you have a Delete button, I recommend clicking the Remove Reference Only button. Clicking the Remove Reference Only button and using the Finder to move the file to the Trash takes much less time than recovering a file you permanently deleted by mistake.

Renaming a Project

Naming your project is an important decision because Xcode uses the project name as the application name for application projects. If you want to change the name of your application, the easiest thing to do is rename the project.

Renaming a project is more involved than changing its filename in the Finder. Select the project in the project navigator and open its file inspector by choosing View > Utilities > Show File Inspector. At the top of the file inspector is the Project Name text field. Enter a new name and press the Return key. A sheet opens asking if you want to rename the project content items, which include the target name, the build product name, the precompiled header file, and the `Info.plist` file. There are checkboxes to control what gets renamed. Click the Rename button to finish renaming the project. If you don't want to rename the project, click the Don't Rename button.

Modernizing a Project

When you open a project, Xcode evaluates it to make sure it conforms to the latest standards. Open the issue navigator to see if anything needs to be updated. If Xcode finds any outdated settings, they appear as a Project Modernization warning in the issue navigator. You can also choose Editor > Validate Settings to check for outdated settings.

Selecting the Project Modernization warning from the issue navigator or choosing Editor > Validate Settings opens a sheet with a list of modernization issues. Selecting the checkbox next to an issue tells Xcode to make the change when you click the Perform Changes button. Click the Don't Perform Changes button to keep the existing project settings.

What does Xcode check when it looks for modernization issues? Xcode checks for outdated build settings, file formats, and SDKs. Checking for outdated SDKs makes working with old Xcode projects easier in Xcode 4. If you have ever opened an old Xcode project and been unable to build it due to a missing SDK, you will appreciate Xcode's modernizing capabilities.

Workspaces

A workspace groups multiple projects in one window. Xcode treats the workspace as a single unit. All the projects in the workspace share the same build directory. Files in one project are visible to the other projects in the workspace. Use a workspace if you have a group of related projects. Suppose you have one project that builds a library that a second project needs. Place both projects in a workspace and Xcode will build the projects in the proper order.

When working with workspaces keep in mind that a workspace is a container for projects. The projects exist outside of the workspace. You can place a project in multiple workspaces. You can remove a project from a workspace without affecting the project or other workspaces.

Creating a Workspace

There are two ways to create a workspace. First, you can create a workspace out of an existing project by opening the project and choosing File > Save As Workspace. Second, choose File > New > Workspace to create an empty workspace. I recommend giving the workspace a different name than your project to avoid confusion.

When you create a workspace, you'll notice it looks a lot like Xcode's project window. The only visual difference between a workspace window and a project window is a workspace can contain multiple projects.

Adding Projects to a Workspace

There are three ways to add a project to a workspace. First, drag the project to the workspace. Second, choose File > Add Files to WorkspaceName. Third, right-click in the project navigator and choose Add Files to WorkspaceName.

When adding a project to a workspace, make sure you have not selected a project in the project navigator. If you select a project, the project you're trying to add to the workspace gets added to the selected project, not the workspace. The safest way to add a project to the workspace is the third way. Right-click in the project navigator below the projects and choose Add Files to WorkspaceName.

Organizer

The Organizer is an auxiliary window to perform tasks that aren't tied to a particular project. Use the Organizer to do things you can't do from the project window.

Opening the Organizer

Choose Window > Organizer or click the Organizer button in the project window's toolbar to open the Organizer. At the top of the Organizer are five buttons that correspond to the five sections of the Organizer: Devices, Repositories, Projects, Archives, and Documentation. The Devices section is for iOS developers. Skip to the section "Organizer for iOS Applications"

to learn more about what you can do when you click the Devices button. If you're a member of Apple's paid Mac developer program, clicking the Devices button allows you to add your Mac to the paid developer program's portal.

The Repositories section shows your version control repositories. Use the repositories window to add new repositories and examine your commit messages. Read Chapter 8, "Version Control", for more information on using version control in Xcode.

The Projects section contains a list of recently opened projects. Use this section to open projects, remove projects from the Organizer, examine snapshots, and delete derived data, like build products and object files. Select a project from the list on the left side of the Organizer to access the project's snapshots and derived data. Right-click the selected project to open the project, reveal the project in the Finder, or remove the project from the Organizer.

The Archives section shows your archived applications. Archive each version of a shipping application so you can test that particular version for bugs that users report. The Archived Applications section lists the project builds you archived when choosing Product > Archive. An archived application is a compressed archive of an iOS or Mac application, similar to a zip archive. From this section you can validate your application to make sure it is ready to submit to the App Store, share copies of the application with testers, and submit the application to the App Store. When sharing a Mac application, you can share it as a Mac App Store package, an application package, or as an archive. You will be asked for a save location. When sharing an iOS application, you can share it as an App Store package or as an archive.

The Documentation section gives you a documentation window to read developer documentation. The section "Reading Developer Documentation" in Chapter 2, "Editing Source Code", contains detailed information on reading documentation.

Organizer for iOS Applications

When you click the Devices button at the top of the Organizer, you will see two sections on the left side of the Organizer: Library and Devices. The Library section has the items Provisioning Profiles, Software Images, Device Logs, and Screenshots. Older versions of Xcode may have a Developer Profile section. The Devices section has the items Provisioning Profiles, Applications, Console, Device Logs, and Screenshots. If you haven't connected a device to your Mac, you will not see the Provisioning Profiles, Applications, and Console items.

After reading the previous paragraph you noticed both the Library and Devices sections have Provisioning Profiles, Device Logs, and Screenshots items. The Devices section stores the provisioning profiles, device logs, and screenshots for a single device while the Library section stores all profiles, logs, and screenshots.

Developer Profile

If your version of Xcode has a Developer Profile section, it shows a list of your identities and provisioning profiles. The main use of the developer profile is to transfer your iOS developer information to a new Mac. Export your developer profile to a file, copy the file to the new Mac, and import the profile on the new Mac.

Newer versions of Xcode place this information in a Teams section or in the Provisioning Profiles section.

Provisioning Profiles

Clicking the Provisioning Profiles item shows a list of your provisioning profiles. The Organizer displays the name of each profile and the expiration date. The provisioning profile tells the device to accept applications that you signed. To add a provisioning profile to your device, connect the device to your Mac and drag the provisioning profile to the device's icon in the Organizer.

Click the New button at the bottom of the Organizer to add a new provisioning profile. Select a provisioning profile from the profile list and click the Refresh button to redownload certificates and update the provisioning profile.

Software Images

When you upgrade your device to the latest version of iOS, a software image for the latest version is copied to your Mac's hard drive. Selecting Software Images lets you view the images that have been copied to the hard drive and delete them if you don't want them on your hard drive.

Device Logs

Clicking the Device Logs item shows a list of crash logs and other logs for your iOS devices. The list tells you the application that created the log, what caused the log entry (examples include crashes and low memory conditions), and the date it occurred. Selecting a log from the list displays it in the Organizer.

If your iOS application crashes, drag the crash log to the Device Logs section of the Organizer. Xcode will add symbols to the crash log to make debugging the crash easier.

In Xcode 4.5 and later the bottom of the Organizer has buttons to import, export, and resymbolicate crash logs. Select a crash log from the list to export or resymbolicate the log.

Screenshots

The Screenshots section shows screenshots of your iOS applications. The Organizer has two panes for screenshots. The left pane has a list of all the screenshots you've taken. Selecting a screenshot shows a full-size version of it in the right pane. Clicking the Save as Launch Image button makes that screenshot the launch image for a project. When you click the Save as Launch Image button, a window opens with a sheet of open projects. Select a project and click the OK button. When you click the OK button, Xcode adds the screenshot to the project.

To take a screenshot of your device, select Screenshots under Devices. Click the New Screenshot button in the lower right corner of the Organizer. The screenshot appears on the right side. To remove a screenshot, select it from the screenshot list and click the Delete button at the bottom of the Organizer.

Clicking the Export button saves the screenshot as a PNG file. Selecting multiple screenshots and selecting the Compare checkbox lets you view the differences between the screenshots. Use the Tolerance slider to adjust the colors of the differences.

Devices

When you connect a device to your Mac, the device appears in the Devices section of the Organizer and a dot appears next to the device. If the color of the dot is not green, it means the device is connected, but not available for development. Select the device and click the Use for Development button. When you tell Xcode to use the device for development, a green dot appears next to the device.

Selecting the device provides general information about the device, including the iOS version, the provisioning profiles, the installed applications, the number of crash logs, and the number of screenshots. Clicking the focus button next to the provisioning profiles, applications, device logs, or screenshots takes you to that section.

After you set the device for development, you can use the Organizer to add provisioning profiles to the device, add applications to the device, view the console, view crash logs, and take screenshots of the device. To be able to add applications and provisioning profiles to your device, you must join the iOS Developer Program and pay the annual membership fee.

The Devices section has the following entries:

- Provisioning Profiles
- Applications
- Console
- Device Logs
- Screenshots

You must connect a device to your Mac to see the Provisioning Profiles, Applications, and Console entries. The Applications entry shows the applications installed on the device. Use this section to add applications to and remove applications from the device.

The Console entry displays the console log for the device. Click the Save Log As button at the bottom of the Organizer to save the log on your Mac.

The Provisioning Profiles, Device Logs, and Screenshots entries in the Devices section store the provisioning profiles, device logs, and screenshots for a single device. Read the “Provisioning Profiles”, “Device Logs”, and “Screenshots” sections earlier in this chapter for more information on provisioning profiles, device logs, and screenshots.

Chapter 2

Editing Source Code

After creating a project, the next thing you're going to do in Xcode is write source code. This chapter focuses on topics to improve your code writing experience in Xcode.

The Editor Pane

The editor pane is where you write your source code. Selecting a source code file from the project navigator opens the file in the editor. Double-clicking a source code file in the project navigator opens a separate editor window. Figure 2.1 shows a typical editor pane, which contains four parts.

- Jump bar
- Editor
- Gutter
- Focus ribbon

Jump Bar

The jump bar lets you quickly reach a file or a method inside the file. Click on an item in the path control and use the menus to reach a particular file or a method inside a file. If you hold down the Command key when clicking on the path control, the file names or method names appear in alphabetical order.

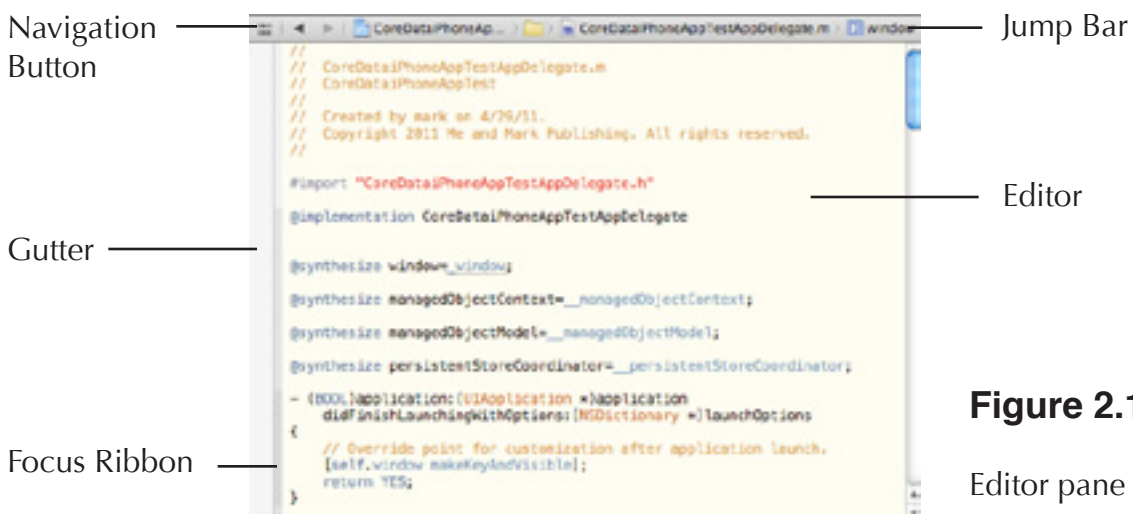


Figure 2.1

Editor pane

On the left side of the jump bar is the navigation button. The navigation button for a source code file has the following menus: Recent Files, Unsaved Files, Counterparts, Superclasses, Subclasses, Siblings, Categories, Protocols, Includes, and Included By. Choosing a file from one of the menus opens the file in the editor. If there are no items to show in a menu, the menu is grayed out.

The Recent Files menu contains any recently opened files in your project. There are also menu items to clear the list and to specify the maximum number of files the Recent Files menu should show.

The Counterparts menu is for C-based languages, where most implementation files have a corresponding header file. The header file is the implementation file's counterpart and vice versa.

The Superclasses menu contains the header files of the class's superclasses. The Subclasses menu contains the header files of the class's subclasses. The Siblings menu contains the header files of the classes that share a superclass with the current file.

The Categories menu contains a list of categories. A category allows you to add methods to an existing class. The Protocols menu contains a list of protocols the current file adopts. A protocol is a list of methods for another class to implement. Many source code files have empty Categories and Protocols menus.

The Includes menu contains a list of header files the current file includes. The Included By menu contains a list of files that include the current file.

Editor and Gutter

The editor is where you edit source code. You've used word processors and text editors before. Xcode's editor isn't radically different from editors you've previously used. The gutter runs along the left edge of the window. Its main use is for setting breakpoints, which I discuss in the section "Setting Breakpoints" in Chapter 7, "Debugging".

Focus Ribbon

The focus ribbon, which is located between the gutter and the editor, lets you highlight and fold blocks of code. Moving the mouse cursor over a block in the focus ribbon highlights the block of code. Clicking the focus ribbon folds the block, which means the code inside the block is hidden in the editor. When you fold a block of code, a tiny triangle appears in the focus ribbon. Clicking the triangle unfolds the block of code in the editor.

Assistant Editor

In the project window toolbar is the Editor group of buttons. Click the center button to switch to the assistant editor. The assistant editor opens a source code file and its counterpart. For C, C++, and Objective-C programmers, the assistant editor allows you to view an implementation file and its header file side by side.

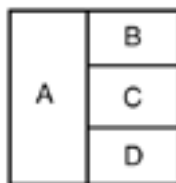
On the right side of the assistant editor's jump bar is the Add Assistant Editor button. Clicking the Add Assistant Editor button creates another editor. The assistant editor is the only editor that allows you to split the editor. Next to the Add Assistant Editor button is the Remove Assistant Editor button. Click it to remove an assistant editor you created. The primary assistant editor (the first editor) does not have buttons to add and remove editors.

The assistant editor initially shows the two editors side by side. If you choose View > Assistant Editor, you can customize the look of the assistant editor. There are four layouts.

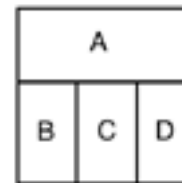
- Assistant editors on right
- Assistant editors on bottom
- All editors stacked horizontally
- All editors stacked vertically

Assistant editors on right is the initial layout. The best way to show the differences in the four layouts is with a picture. Figure 2.2 shows what happens with each layout when you click the Add Assistant Editor button twice to add two editors. Switching to the assistant editor gives you editors A and B. Clicking the Add Assistant Editor button creates editor C, and clicking the Add Assistant Editor button a second time creates editor D.

Assistant Editors on Right



Assistant Editors on Bottom



All Editors Stacked Horizontally



All Editors Stacked Vertically

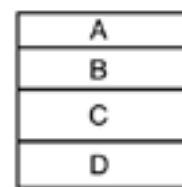


Figure 2.2

Assistant editor layouts

Code Completion

Code completion tells Xcode to finish the names of your program's functions and variable names, saving you from having to type the whole name. Code completion also helps reduce typographical errors in your code.

Xcode's code completion uses a pop-up menu known as a completion list. The completion list contains all the possible matches for what you type. As you type more of the function or variable name, the completion list updates itself to reflect what you type. Select an entry from the completion list and press the Return key to complete the code. If you don't want to complete the code, press the Esc key to close the completion list.

Pressing the Esc key also opens the completion list, but Xcode should be set up for automatic code completion initially. When you start typing, the completion list should appear. If the completion list is not appearing, open Xcode's Text Editing preferences and make sure the Suggest completions while typing checkbox is selected.

When you complete a function that takes arguments, Xcode provides a placeholder token for each argument. Replace the tokens with your own values. Enter a value for the argument and press the Tab key to fill the token and move to the next argument. Pressing the Tab key without entering a value for the token moves you to the next argument without filling the token.

Customizing Code Editing

Programmers have different opinions about what makes code easy to read. Some people prefer black text on a white background while others prefer white text on a black background. Some programmers indent code four spaces, others prefer to indent two spaces, and others prefer to indent eight spaces. Some people want to see line numbers in the editor while others don't.

Xcode has many preferences to customize your experience editing source code. The preferences for customizing source code editing are the following:

- Fonts and Colors
- Text Editing
- Key Bindings

Fonts and Colors Preferences

Xcode gives different areas of your code their own text color to make reading the code easier. The Fonts and Colors preferences is where you set the text color, font, and font size for your code.

Color Themes

Xcode comes with several color themes. Select a theme from the theme list. If you don't like any of Xcode's themes, select the best matching theme, click the + button, and choose Duplicate from the pop-up menu. Enter a name for the theme. Use your theme to change the source code's font and color.

Categories

To the right of the theme list is the category list. Looking at the category list, you will see lots of categories, including comments, class names, language keywords, strings, and numbers. Xcode provides categories for the editor and the console. To change the text color, select a category. Below the category list is a Font text field and a color button. Click the color button to change the color.

To change the font, select a category and click the button on the right edge of the Font text field. When you look at Xcode's color themes, you'll notice each theme uses one font. If you want to change the font, select multiple categories before making the change. Changing the font for each individual category can be tedious.

Setting Colors for Non-Text Items

In addition to setting the text color, you can set the colors for the editor's background, selected text in the editor, the insertion point cursor, and invisible characters. At the bottom of the preferences window are buttons to set the color for the background, selection, cursor, and invisibles. Click the appropriate button to change the color.

Text Editing Preferences

Xcode's Text Editing preferences allow you to customize your Xcode editing experience. The Text Editing preferences have two tabs: Editing and Indentation. Clicking the Editing tab lets you set miscellaneous text editing options. Some options you can set include the following:

- What character is generated when you press the Return key to end a line of code: a carriage return or a line feed. Mac OS (Mac OS 9, which predates Mac OS X), Unix, and Windows end lines differently. Mac OS generates a carriage return. Unix generates a line feed. Windows generates a carriage return and a line feed. Mac OS X uses Unix line endings.
- Showing a page guide that lets you know when a line of code is too long.
- Showing line numbers in the editor.
- Showing the code folding ribbon.
- The file encoding to save files, which is initially 8-bit Unicode.

The next section covers what happens when you click the Indentation tab.

Indentation Preferences

Indenting your source code makes the code easier to read. Xcode's Indentation preferences let you control how Xcode indents your code. Click the Indentation tab in Xcode's Text Editing preferences to access the Indentation preferences. Xcode's Indentation preferences contain three sections: Tabs, Line Wrapping, and Syntax-Aware Indenting.

Tabs

The Tabs section controls what happens when you press the Tab key to indent code. Use the Prefer indent using pop-up menu to tell Xcode to insert spaces or tabs when indenting code. What is the difference between inserting spaces and inserting tabs? Suppose you are indenting four spaces every time you press the Tab key. If you insert spaces, Xcode inserts four space characters. If you insert tabs, Xcode inserts one tab. Assuming you set the tab width to four, the code looks identical in Xcode's editor. The difference shows up if you send the file to someone with a different tab width. If you insert spaces, the code is indented four spaces on the other person's computer. If you insert a tab, the amount of indentation depends on the other person's settings. If he has a tab width of two spaces, the code will be indented two spaces on his computer. Inserting tabs or spaces is a matter of personal preference and a subject of intense debate among programmers.

The Indent width field determines how many spaces Xcode indents code when pressing the Tab key. The Tab width field determines how many spaces Xcode moves ahead when it encounters a tab character. Set the Indent width field if you're going to insert spaces when pressing the Tab key, and set the Tab width field if you're going to insert tab characters.

The Tab key pop-up menu tells Xcode what to do when you press the Tab key. There are three options: indent in leading whitespace, indent always, and insert a tab character. If you're going to use Xcode's syntax-aware indenting, indent in leading whitespace works better than indent always. The syntax-aware indenting keeps you from indenting past a

certain point in a line of code when you indent always, which means pressing the Tab key doesn't move the cursor past a certain point. The certain point depends on your code. For a nested loop the certain point will be farther to the right than the first line of a function. When you indent in leading whitespace, pressing the Tab key always moves the cursor.

The Indentation preferences apply to all projects you create in Xcode and the files in those projects. You can override the tab settings for a project by selecting the project file in the project navigator and going to the Text Settings section in the file inspector. You can override the tab settings for a single file by selecting the file in the project navigator and going to the Text Settings section. Choose View > Utilities > Show File Inspector to open the file inspector.

Line Wrapping

If you're typing a really long line of code that extends past the end of the editor's view, you can have Xcode wrap the line in the editor. You can specify how many spaces to indent the wrapped line. Line wrapping allows you to view a long line of code without having to break the line into multiple lines. Line wrapping helps if you're writing code on a smaller screen, such as a laptop.

Syntax-Aware Indenting

If you select the Syntax-aware indenting checkbox, Xcode turns on automatic code indenting. When you type a statement that calls for indentation, such as an `if`, `for`, or `while` statement, and press the Return key, Xcode indents the new line for you. You can specify the characters that trigger the indenting as well as control how Xcode indents code comments.

Key Bindings

Use the Key Bindings preferences to set keyboard shortcuts for Xcode. Xcode has two kinds of key bindings: menu and text. The menu key bindings let you create a keyboard shortcut for every menu in the Xcode menu bar. The text key bindings let you set keyboard equivalents for things like selecting text, marking text, and moving the cursor in the editor.

Xcode comes with a default set of key bindings, but you can create a custom set of key bindings. To create a custom set of key bindings, select a command set from the list on the left side of the preferences window, click the + button, and choose Duplicate SetName to create a copy of the set. Use the copy of the command set to bind keys.

To add a key binding for a command or menu item, select the item in the table and click the Key column or press the Return key. Enter the keyboard shortcut you want to use. If the shortcut you want to use is being used by another command, a message appears at the bottom of the window telling you what the key combination is currently bound to. Select another item in the table to change the key binding, overwriting the previous key binding with the key binding you just added. Click the minus button to keep the existing key binding, which means you have to pick a new key combination for the selected command.

The last half of the previous paragraph is a little confusing. Let me use an example to illustrate key binding conflicts. Suppose you want to use the key combination Control-A to select a word in the editor. Xcode's default command set uses Control-A to move to the beginning of a paragraph. You select the Select Word command and enter Control-A as the shortcut. Xcode reports that Control-A is currently bound to Move to Beginning of Paragraph. Selecting another item in the table changes the Control-A key combination to the Select Word command. Clicking the minus button in the Key column for Select Word keeps the Control-A combination for the Move to Beginning of Paragraph command.

To remove a key binding, select it, click the Key column or press the Return key, and click the minus button.

Use the search field to look for a particular command or to check if a key combination is being used as a shortcut for a command. The icon button in the search field lets you customize the search. If you want to see what commands have the P key in their keyboard shortcut, click the icon button, choose Key Equivalent, and enter P in the search field.

Code Snippets

Code snippets keep you from having to type the same code over and over. Xcode comes with a bunch of code snippets you can use in your programs, and you can create your own snippets.

Using a Code Snippet

Choose View > Utilities > Show Code Snippet Library to open the code snippet library in the lower right corner of the project window. To use a code snippet, select it from the code snippet library and drag it to the editor.

Initially the code snippet library shows all code snippets. Use the search field at the bottom of the library to find a particular snippet. Use the Code Snippet Library pop-up menu to filter the snippets the code snippet library displays. The code snippet library has the following filters: display only Apple's iOS snippets, display only Apple's Mac snippets, and display only user-defined snippets.

Creating a Code Snippet

To create your own code snippet, select the text that's going to compose the snippet and drag it to the code snippet library. A pop-up editor opens where you can name the snippet, add a summary, add a completion shortcut, add a completion scope, edit the contents of the snippet, and specify a platform and language. Click the Done button to finish creating the snippet.

Choosing C as the language allows your snippet to be invoked for C, C++, and Objective-C source code files.

You can delete a custom code snippet by selecting it from the code snippet library and pressing the Delete key. You can't delete the code snippets that ship with Xcode.

Completion Shortcuts

A completion shortcut lets you invoke the snippet by typing a phrase. If you give a snippet the shortcut `xyz`, when you enter `xyz` in the editor, a completion list opens. You can give multiple snippets the same completion shortcut. Choose the snippet you want to use from the completion list. If you don't want to use a snippet, press the Esc key.

Completion Scope

A completion scope determines when a code snippet should be suggested as part of Xcode's code completion. You must have a completion shortcut to use the completion scope.

Xcode initially gives a snippet the completion scope All, which means the snippet always appears in the completion list when you enter the completion shortcut. You can control when the snippet appears in the completion list by using the Completion Scopes pop-up menu. Your completion scope options depend on the language you specify when creating the code snippet, but the following options are commonly available:

- Class Implementation, which means you must be inside the `@implementation` section and outside a method for the code snippet to appear in the completion list. Only Objective-C snippets have this option.
- Class Interface Methods, which means you must be inside the list of class methods in a header file for the code snippet to appear in the completion list. Only Objective-C snippets have this option.
- Class Interface Variables, which means you must be inside the list of class variables in a header file for the code snippet to appear in the completion list. Only Objective-C snippets have this option.
- Code Expression, which means you must be in a code expression for the code snippet to appear in the completion list. In a C-based language, you must be inside a pair of braces to be in a code expression.
- Function or Method, which means you must be inside a function or method for the code snippet to appear in the completion list.
- Preprocessor Directive, which means you must be inside a `#define` statement for the code snippet to appear in the completion list.
- String or Comment, which means you must be inside a string or a code comment for the code snippet to appear in the completion list.
- Top Level, which means you must be outside all code blocks for the code snippet to appear in the completion list. In a C-based language, to be outside all code blocks means you have to be outside a pair of braces.

If you want the snippet to appear in multiple completion scopes, but not all the time, click the + button to add another completion scope.

Placing Tokens in Your Code Snippets

Some of Apple's code snippets have tokens, where you can insert variables into a snippet. To add a token to a code snippet, use the following tag:

```
<#TokenName#>
```

Suppose your snippet calls the OpenGL function `glColor3f()`. The following code creates tokens for the color's components:

```
glColor3f(<#red#>, <#green#>, <#blue#>);
```

When you use the snippet, you'll be able to enter your own values for `red`, `green`, and `blue`. Use the Tab key to cycle through the tokens.

Examining a Code Snippet

Select a snippet from the code snippet library. A pop-up editor opens where you can see the code that comprises the snippet. You can modify snippets you create by clicking the Edit button. Click the Done button when you're finished examining or editing a snippet.

Tab Bar

The tab bar supports tabbed editor windows, similar to what web browsers like Safari and Firefox have. The tab bar is initially invisible. To show the tab bar, choose View > Show Tab Bar. Click the + button on the right side of the tab bar to add a new tab. Double-click the tab name to rename the tab.

Drag a tab off the tab bar to create a separate window for the tab.

Refactoring Tools

Refactoring takes existing source code and improves it without changing the code's behavior. Why would you want to refactor your code? Suppose you've written an application that works well, but the code is difficult to understand in some places. You want to make the code easier to understand, but you don't want your changes to break the application. This is the problem refactoring solves. Refactoring lets you make the code easier to understand without breaking it.

Xcode provides tools to help you refactor your code. Xcode's refactoring tools work with C and Objective-C programs. They do not work with C++ and Objective-C++ code.

To refactor a piece of code, select the code you want to refactor in Xcode's editor window and choose Edit > Refactor. There are six transformations you can perform on your code.

- Rename, which changes the name of a class, variable, or method.
- Extract, which creates a new method out of a piece of code.
- Create superclass, which creates a superclass for an existing class. It can also create .h and .m files for the new class. To create a superclass, select the class name from the header file.
- Move up, which moves the declaration and definition of a method to its superclass.
- Move down, which moves the declaration and definition of a method to a subclass.
- Encapsulate, which creates accessors out of a variable. It also modifies the code that accesses the variable to use the accessors.

The transformations that are available to you depend on the code you select. If you select a variable name, the Extract transformation will be unavailable. If you select ten lines of code, the Rename transformation will be unavailable.

When you choose a transformation, a sheet opens for you to make the changes to the selected code. What appears in the sheet depends on the transformation you choose. If you choose to rename a variable, the sheet contains a text field for you to enter the new variable name. Click the Preview button to move to the next step in the refactoring process.

Clicking the Preview button lets you see the changes that will occur in your code. A list of files affected by the refactoring will appear after clicking the Preview button. Selecting a file shows you the changes that will occur in the file. There is a checkbox next to each file. Each checkbox is initially selected, which means Xcode will apply the refactoring to each of the files. If you deselect the checkbox next to a file, Xcode will not apply the changes to that file.

Click the Save button to do the refactoring. Click the Cancel button to stop the refactoring.

When you click the Save button, a sheet opens asking if you want Xcode to take automatic snapshots before refactoring. Turning on automatic snapshots tells Xcode to take a snapshot of your project every time you refactor some code in the project. Xcode takes the snapshot before refactoring. If you don't like the changes you made, you can go back to the snapshot. Click the Enable button if you want Xcode to take snapshots each time you refactor your code.

Converting Your Project to ARC

Xcode also has a project-wide refactoring: converting Objective-C code to ARC. ARC stands for Automatic Reference Counting, which provides automatic memory management of Objective-C objects. The refactoring converts an existing project to use ARC.

Choose Edit > Refactor > Convert to Objective-C ARC to convert your project. ARC does not work with garbage collection on Xcode 4.2. If your project uses garbage collection, you must turn it off by setting the Objective-C Garbage Collection build setting to Unsupported.

When you refactor your project to use ARC, a sheet opens with a list of your project's targets. Select the checkbox next to a target to refactor that target. Selecting the checkbox tells Xcode to refactor all the source code files in the target. Click the disclosure triangle next to a target to see a list of source code files. Deselect a checkbox to tell Xcode not to refactor a file. Click the Check button to start the conversion.

After clicking the Check button, click the Next button to open a preview window that shows the changes Xcode will make when you refactor the project. The preview window contains a list of files affected by the refactoring. Selecting a file from the list shows you the changes

that will occur in the file. Each file in the list has a checkbox next to it. The checkbox is initially selected, which means Xcode will apply the refactoring to all the files in the list. If you deselect the checkbox next to a file, Xcode will not apply the changes to that file.

Click the Save button to do the refactoring. When you convert the code to ARC, Xcode takes a snapshot of the project so you can go back if the conversion causes problems in your application. Click the Cancel button to stop the refactoring.

Converting to Modern Objective-C Syntax

Xcode 4.4 adds a second project-wide refactoring: converting Objective-C code to modern Objective-C syntax. Modern Objective-C syntax falls into two main categories. The first category is increased support for literals using the @ character. If you have worked with `NSString` objects, you know you can use the @ character to define a string literal.

```
NSString* myString = @"Hello";
```

The increased support adds support for literals to `NSNumber`, `NSArray`, and `NSDictionary` objects, which makes initializing numbers, arrays, and dictionaries easier.

```
NSNumber* myNumber = @10;
NSNumber* myDouble = @3.25;
NSArray* myArray = @[ @10, @15, @20 ];
NSDictionary* myDictionary = { @1: @"red", @2: @"green", @3:
    @"blue", @4: @"alpha" };
```

The second category is support for using C array syntax to access elements in `NSArray` and `NSDictionary` objects. To access element 3 in an `NSArray` with the old syntax, you would use the following code:

```
[myArray objectAtIndex:3];
```

With modern Objective-C syntax you can access element 3 with the following code:

```
myArray[3];
```

Choose Edit > Refactor > Convert to Modern Objective-C Syntax to convert your project. Xcode walks you through the conversion. The first step is to tell you any changes that will be made to your project settings. Click the Next button to move on. The second step is to choose the targets to convert. Click the Next button to generate a preview of what will change. There are two source code views. The left view shows how the code will look after the conversion. The right view shows the original code. Between the two views is a button

with a number. If you don't want to make a change, click the button and choose Discard Change. Click the Save button to do the conversion. Click the Cancel button to stop the conversion.

Fix-it

Fix-it checks your syntax as you type, highlighting any syntax errors in your code. When a syntax error occurs, an error icon appears in the gutter on the left side of the editor. Clicking the icon opens a pop-up editor with the compiler error message. Below the error message is the suggested fix for the error. Press the Return key to apply the suggested fix. Press the Esc key to close the window without applying the fix.

Not every error has a suggested fix. If Xcode cannot provide a suggested fix, no pop-up editor opens when you click the error icon.

You must be using the LLVM compiler to use Fix-it. Fix-it works with C, C++, Objective-C, and Objective-C++ code.

Reading Developer Documentation

When you're writing source code that uses Apple's technologies, you occasionally need to consult Apple's developer documentation. You don't have to leave Xcode to read developer documentation. Open the Organizer and click the Documentation button at the top of the Organizer. You can also choose Help > Documentation and API Reference to open Xcode's documentation window. The documentation window has the following elements:

- A navigator where you can browse documentation, search it, and access your bookmarks.
- A jump bar, which lets you reach an area of a document quickly.
- The documentation area, which is where you read the documentation.

If you open Xcode's Help menu, you will notice that it contains a search field. The search field allows you to search Apple's Developer Tools documentation, but does not search Apple's Mac and iOS documentation. Enter a search term to find help topics in the Developer Tools documentation.

Browsing Documentation

The Organizer's navigator runs along the left side of the window. At the top of the navigator is a selector bar with three buttons. Click the left button to browse the documentation. The navigator has one entry for each documentation set you've installed. Most of you have at least three documentation sets installed: the Mac OS X set, the iOS set, and the Developer Tools set.

Selecting a documentation set in the navigator takes you to the base page for that set. Apple groups Mac OS X and iOS documentation three ways: by resource type, by topic, and by framework. On the left side of the documentation area is a list of category links containing the available resource types, topics, and frameworks. Click a category link to show the documents for that category. Figure 2.3 provides an example of what the documentation window looks like after clicking a link. The Xcode documentation set has no list of links. It contains a list of all documents on Apple's developer tools.

Some categories contain a lot of documents. If you click the Sample Code link under Resource Types in the Mac OS X documentation, you will find hundreds of entries. Use the search field above the list of documents to filter what appears in the documentation window.

Each documentation set has a disclosure triangle next to it. Clicking it opens up subsets of the documentation. Using the jump bar to reach a specific piece of documentation is usually faster than clicking disclosure triangles in the navigator.

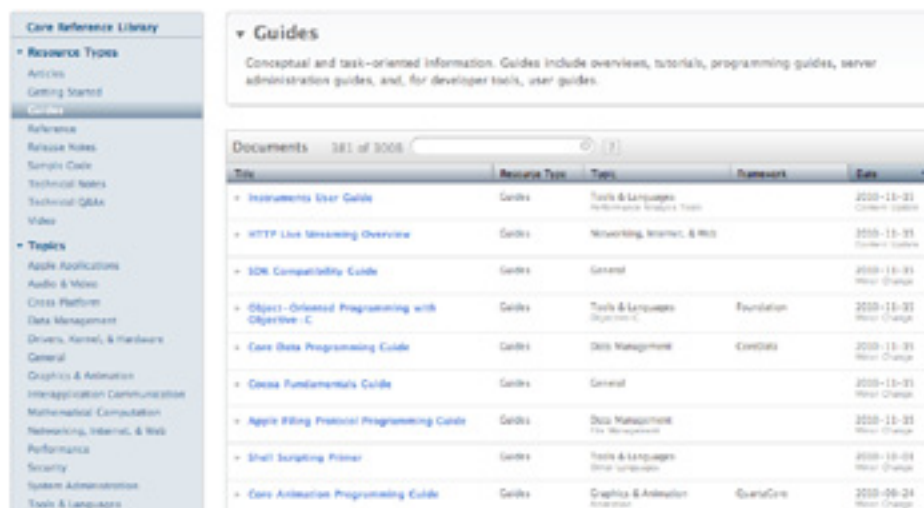


Figure 2.3

Documentation window during browsing

Searching Documentation

Searching the documentation is pretty simple. Click the center button in the navigator selector bar. Enter what you want to search for in the search field in the navigator and press the Return key. The search results appear on the left side of the documentation window, divided by documentation type: reference, guides, and sample code. Figure 2.4 shows an example of the search results.

To open more search options, click the icon button on the left side of the search field and choose Show Find Options. There are three find options. The first option is match type that has three possible values: contain search term, start with search term, and match search term. The initial value is contain search term, which means the search results show every document that contains the search term. If you choose start with search term, the search results show every document that starts with the search term. If you choose match search term, the search results show every document that is an exact match for the search term. Choosing match search term reduces the search results greatly.

The second option lets you control the documentation sets to search. Initially Xcode searches all document sets. If you're an iOS developer, you may want to limit searches to the iOS documentation. Use the pop-up menu to specify the documentation sets to search. The third option controls the programming languages to search: C, C++, JavaScript, and Objective-C. Initially Xcode searches for documentation in all four languages. Use the pop-up menu to limit the languages to search.

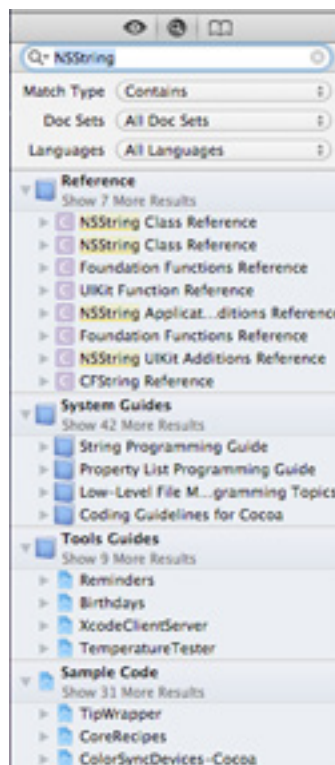


Figure 2.4

Documentation
search results

Bookmarks

Bookmarks let you quickly reach documents so you don't have to waste time navigating the documentation. To bookmark the page you're reading, right-click in the documentation area and choose Add Bookmark for Current Page.

The bookmarks button is the right button in the navigator selector bar. Clicking the bookmarks button fills the navigator with a list of all the documents you bookmarked. Select a document from the list to read it.

You can rearrange bookmarks by dragging them in the navigator. To remove a bookmark, select it in the navigator and press the Delete key.

Quick Help

Xcode's documentation window is fine for reading longer pieces of documentation, but when you want a quick explanation of a class or method, there's Quick Help. Quick Help works best with the Cocoa, Cocoa Touch, and Core Foundation frameworks. It also helps when you're examining code examples you haven't written.

Invoking Quick Help

There are two ways to invoke Quick Help. First, choosing View > Utilities > Show Quick Help Inspector opens Quick Help on the right side of the project window. Moving the cursor inside a symbol in the editor fills the Quick Help section with information about that symbol. Examples of symbols are classes, functions, and variables.

Using the first option keeps the utility area open all the time. If you want to hide the utility area to free up screen space but still take advantage of Quick Help, there is a second option. Option-clicking a symbol in the editor opens a Quick Help pop-up editor. Command-clicking a symbol opens the header file where the symbol resides. Command-option-clicking a symbol opens the header file in the Assistant Editor so you can view both your code and the header file.

Starting with Xcode 4.4 you can view Quick Help from Xcode's code completion list. Select an entry from the completion list. Quick Help for the selected entry appears at the bottom of the completion list. Clicking the More link takes you to the selected entry's documentation in Xcode's documentation window.

What Quick Help Displays

When you select a symbol from the editor, Quick Help provides the following information for the symbol:

- The name of the symbol. If the symbol has a link, clicking the link opens the Organizer and displays the documentation for the symbol.
- Declaration. Most classes won't have a declaration. The Declaration section for methods shows the return type and the arguments the method takes. The Declaration section for data structures shows the members of the data structure.
- Availability, which tells you the versions of Mac OS X or iOS that can use the symbol.
- Description, which is the description of the symbol.
- Parameters, which describes each function argument.
- Return values, which describes what the function returns. Not all functions have information on parameters and return values.
- Declared, which contains a link to the header file where the symbol is declared.
- Reference, which contains a link to reference material on the symbol.
- Related API, which contains links to related methods. Many entries won't have related API.
- Related documents, which contains links to any additional documentation about the symbol.
- Samples, which contains links to sample code that uses the symbol.

The information Quick Help provides for a symbol depends on the symbol. Quick Help provides more information for Cocoa and Cocoa Touch classes than it provides for your application's symbols. Xcode uses the documentation sets you installed to display Quick Help information. This means symbols from installed documentation sets have more information for Quick Help to display than other symbols. Read the "Installing Third-Party Documentation" section later in this chapter for information on installing documentation from sources besides Apple.

Updating Documentation

Xcode's documentation is constantly changing so the documentation that was installed when you installed Xcode may not be up-to-date. Staying current is not too difficult as long as you have a broadband Internet connection.

Open Xcode's Downloads preferences and click the Documentation tab to open Xcode's Documentation preferences. Click the Check and Install button to update the documentation. If you want Xcode to automatically install updates for you, select the Check for and install updates automatically checkbox.

Underneath the Check and Install button is a list of available documentation sets. Xcode initially installs the Xcode library, the Mac OS X library for the version of Mac OS X you're running, and the iOS library for the iOS SDK you've installed. Documentation sets you have not installed have an Install button (or Get button) next to them. Click the Install button to install that set.

Installing Third-Party Documentation

Xcode's documentation viewer is not limited to Apple's documentation. If you have third-party documentation you want to install, click the + button at the bottom of the preferences window. The documentation must be an Atom or RSS feed. Atom feeds start with `http://` while RSS feeds start with `feed://`. Use the Feed URL text field to enter the location of the feed you want to install.

After adding a documentation feed, click the Install button to install it. The documentation set appears in the list of sets when you click the Browse button. You can find third-party documentation sets you've installed in the following location:

```
/Library/Developer/Documentation/DocSets
```

Removing Documentation Sets

Removing a documentation set is not intuitive. There is a minus button at the bottom of the preferences window, but it is always disabled, even for third-party documentation. To remove a documentation set, select it from the preferences window and click the button to the left of the + button at the bottom of the window. Clicking the button opens an information area for the documentation set. Click the Installed Location link to open the documentation set's location in the Finder. Drag the documentation set to the Trash.

Chapter 3

Creating User Interfaces for Mac Applications

The user interface is an integral part of a great Mac OS X application. Cocoa developers use Interface Builder to build their user interfaces. Xcode 4 integrates Interface Builder with Xcode, which allows you to build your user interface without leaving Xcode. This chapter shows you how to use Interface Builder to build user interfaces for Mac applications that use the Cocoa framework.

Starting with Interface Builder

When you create a Cocoa application project, Xcode includes a xib file with the project. For a Cocoa application, the file's name is `MainMenu.xib`. A document-based Cocoa application has a second xib file called `MyDocument.xib`, where `MyDocument` is the name of your `NSDocument` subclass. In a document-based application, `MainMenu.xib` contains the menu bar and `MyDocument.xib` contains the window that will be created when your application opens a new document window.

Select a xib file from the project navigator to open the xib file and start building your user interface. If you can't find your xib file, enter `xib` in the search field at the bottom of the project navigator to show all the xib files in your project.

When you select a xib file from the project navigator, the editor shows the user interface area. There are three sections that make up the user interface area, which you can see in Figure 3.1. On the left is the object list. Click the small button in the lower left corner of the canvas to toggle the icon and hierarchical views of the object list. Next to the object list is the canvas, which is where you lay out the user interface. On the right is the utility area. If the utility area isn't open, open it by clicking the right button in the View group on the right side of the project window toolbar. The utility area contains the user interface elements and inspectors. The user interface elements are in the object library at the bottom of the utility area. If you don't see the object library, choose `View > Utilities > Show Object Library`.

Above the canvas is the jump bar. The jump bar allows you to quickly select an object in the xib file. The jump bar helps you select a user interface element that is nested in a deep hierarchy.

Creating the User Interface

Before you create your interface, make sure the utility area is open and the object library is visible. The utility area contains the object library, which contains the elements you use to build the user interface. Choose View > Utilities > Show Object Library to show the object library, which you can see in Figure 3.2, at the bottom of the utility area.

To lay out the interface, select an element from the object library and drag it. Where you drag the element depends on the element. Elements that will be visible to users of your application, such as views, tables, and buttons, should be dragged to the window where you want them to appear. Controllers and windows should be dragged to the canvas. Cells should be dragged to the table they will reside in. Formatters should be dragged to a text field or table column.

When placing user interface elements in a window, Interface Builder provides guide lines to make sure elements line up with each other and to keep elements from being too close to the edges of the window.

Modifying the Interface

When you lay out your program's user interface, the interface isn't going to look right initially. Buttons, checkboxes, and radio buttons display placeholder text. The initial size of views you drag to the window is pretty small. You'll want to make them bigger. In this section you'll learn how to make changes to your interface.

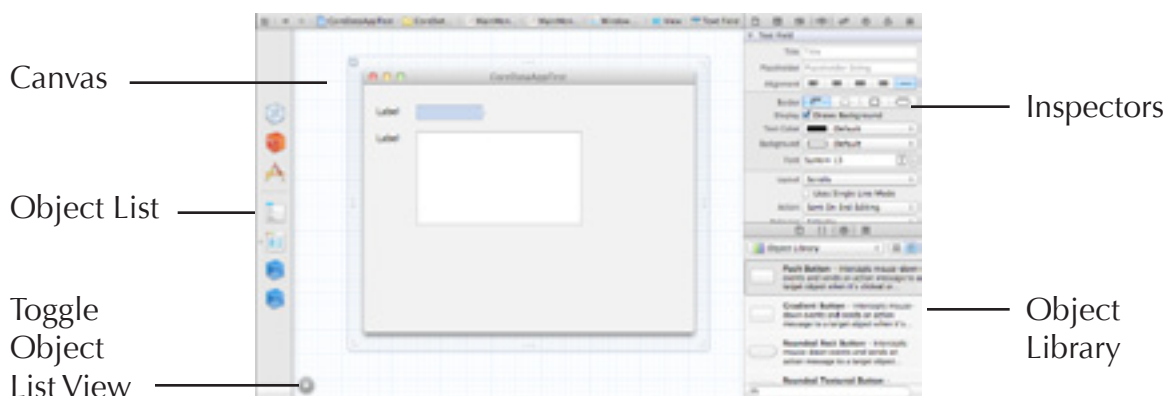


Figure 3.1

User interface area

Selecting an Element

Before you can move, resize, or remove an element, you must first select it. Click on an item in the window to select it. If you want to select multiple elements, command-click the additional elements.

To select the window, click the clear area outside of the window's view. The outline of the window is blue when the window is selected, which you can see in Figure 3.3.

Selecting an Element in a Hierarchy

Sometimes selecting the element you want can be difficult, especially if it is inside a deeply nested hierarchy. Suppose you have an outline view inside a split view. You try to select the outline view to make changes to it, but when you select it in the window, you get the outline view's scroll view. How do you get the outline view?

Use the jump bar, which is above the canvas. The jump bar shows the hierarchy of items in the user interface. Click on an item from the jump bar and use the submenus to get to the element you want to select.

Moving and Resizing Elements

Dragging the selected item changes its location in the window. To resize a control or view, click one of the eight dots bordering the selected control, hold down the mouse button, and move the mouse.

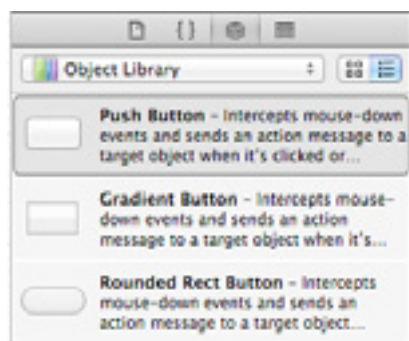


Figure 3.2

Object library

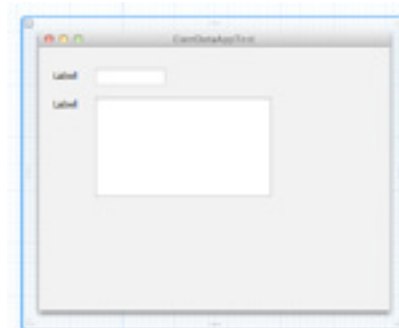


Figure 3.3

A selected window

Deleting an Element

Select the element and press the Delete key to delete it.

Changing the Text of Titles and Labels

Many user interface elements have titles and labels, including buttons, menus, checkboxes, and radio buttons. These elements have placeholder text you don't want in your application. Changing the text is easy in Interface Builder. Double-click the text you want to change and enter the new text.

Making Other Modifications

Use the inspectors at the top of the utility area.

Making Connections

Cocoa programs use messages to send data from one object to another. Connections let your user interface elements send messages to each other, send messages to models and controllers, and receive messages from models and controllers. One of Interface Builder's most powerful features is the ability to make connections between objects without having to write code.

A connection involves two objects: the sender and the receiver. To make a connection, right-click the sender. Drag to the receiver. A HUD window opens with a list of the receiver's outlets and actions. Select an outlet or action from the HUD window.

To remove a connection you previously made, open the connections inspector by choosing View > Utilities > Show Connections Inspector. The connections inspector shows the connections an object has. Figure 3.4 shows an example of a connection. Select the connection and click the X button to remove it.



Figure 3.4

A connection in the connections inspector

Testing the Interface

Interface Builder lets you test your interface without having to build your Xcode project. Choose Editor > Simulate Document to test the interface in the Cocoa Simulator application. Quit the simulator to stop testing and go back to editing your interface.

Creating a Xib File

Add a new xib file to your Xcode project by choosing File > New > File or by dragging a xib file from the file template library to the project navigator. If you choose File > New > File, you can find the xib files for Mac applications in the User Interface section under Mac OS X.

Cocoa applications have five xib files to choose from: application, empty, main menu, view, and window. Because Xcode's Cocoa application project templates include a xib file with a window and menu, you normally don't need to create an application or main menu xib file, but they are available if you want to create a test interface.

Object List

To the left of the canvas is the object list. As the name suggests, the object list contains a list of the xib file's objects. The object list has two views: icon view and hierarchical view. The icon view has icons for each object. The hierarchical view lets you see the names of the objects. The hierarchical view is easier to read, but the icon view takes up less space. Click the small button in the lower left corner of the canvas to change views.

The object list groups the objects into placeholders and objects. The hierarchical view has separate sections for placeholders and objects. The icon view uses a line to separate the placeholders and objects. The most common placeholders are File's Owner, First Responder, and Application. The Objects section contains the objects you add to the xib file, such as windows, menus, and controllers.

File's Owner

The File's Owner is the object that owns the xib file. The object doesn't exist until your application launches so you have no control over the object itself, but you can specify the class of the File's Owner.

1. Select the File's Owner object from the object list.
2. Open the identity inspector by choosing View > Utilities > Show Identity Inspector.
3. Choose a class from the Class combo box.

For the `MainMenu.xib` file Xcode creates for all Cocoa application projects, the default File's Owner is the `NSApplication` class. For document-based applications, the default File's Owner of the document's xib file is the subclass of `NSDocument` Xcode added when you created the project.

First Responder

The First Responder object represents the first responder, which is the first item to receive events the application generates. Normally the user decides the first responder. Suppose a window has a text field where the user types in text. If the user clicks on the text field with the mouse, the text field becomes the first responder. When the user starts typing, the text field handles the keyboard events and fills the field with the text the user types.

Cocoa windows have an `initialFirstResponder` outlet that lets you specify the initial first responder. Let's say you have a window where the user types in personal information such as name, address, and phone number. Each piece of information has its own text field for the user to enter the information. In this case you don't want the user to have to explicitly set the first responder by selecting a text field with the mouse. The user should be able to enter his or her name immediately. To let the user start typing the name, set the initial first responder to the text field where the user enters the name.

1. Make a connection from the window to the text field.
2. Set the outlet to `initialFirstResponder`.

Application

The Application object is a placeholder for your application. Use the Application object to make connections to the application in a document-based application's document xib file. Because the File's Owner is initially set to be the application for `MainMenu.xib`, you usually make connections to the File's Owner instead of the Application object in `MainMenu.xib`.

Object Library

Clicking the Objects button in the library selector bar opens the object library, which gives you access to each user interface element. Selecting an element from the object library opens a pop-up editor that provides a detailed description of the element. Click the Done button when you're finished viewing the description.

The object library initially shows every user interface element. Use the pop-up menu to show a subset of elements. I will explain the groups and the elements in each group shortly. Use the search field to find a particular element.

For more information on user interface elements, read Apple's *OS X Human Interface Guidelines*. The guidelines describe user interface elements as well as provide advice on when to use certain elements. The guidelines are part of Apple's documentation, which you can read in the Organizer.

Controls

The Controls group contains controls. Xcode places many user interface elements in this group. To make covering the elements in the Controls group more manageable, I have divided the elements into the following categories:

- Buttons
- Text controls
- Miscellaneous controls
- Formatters

Buttons

The Controls group contains all the buttons you can add to your interface. You can choose from the following buttons:

- Push button
- Gradient button
- Rounded rectangle button
- Recessed button
- Textured button
- Rounded textured button
- Square button
- Bevel button
- Round button
- Disclosure triangle

- Disclosure button
- Help button
- Inline button

Mail uses an inline button to display the number of unread messages in each mailbox.

Apple's *OS X Human Interface Guidelines* has a section on buttons that explains when to use each type of button. The guidelines are part of Apple's documentation, which you can read from the Organizer.

Text Controls

The Controls group contains the following text controls:

- Label, which displays static text.
- Text field, which you use to enter small amounts of text.
- Secure text field, which lets the user type secure information, such as passwords.
- Search field, which lets the user search in your application.
- Token field.
- Wrapping text field, which is a multiline text field.
- Wrapping label, which is a multiline label.
- Text view, where the user enters large amounts of text. A text editor would use a text view for the user to type documents.
- Form, which is a group of text fields.
- Text field with number formatter, which you would use to let the user enter numerical data.

A token field creates a token out of text. When you enter the email address of someone in your address book in Mail, it creates a token with the person's name.

Use forms instead of individual text fields when you have groups of related text items. Suppose your program requires the user to enter contact information: name, address, phone number, and email address. Instead of creating a text field for each piece of contact information, create one form for all the contact information. Forms make working with multiple text fields easier.

Miscellaneous Controls

The controls I mention in this section are controls that don't fit into one of the other categories. The Controls group contains the following miscellaneous controls:

- Date picker.
- Combo box, which combines a text field with a list.
- Stepper, which increments or decrement a value.
- Checkbox.
- Pop-up button, which contains a pop-up menu.
- Segmented control.
- Radio group, which contains a group of radio buttons.
- Image well.
- Color well, which shows the current color and lets the user choose a color.
- Path control.
- Sliders, which display a range of values.
- Progress indicators, which show the progress of lengthy tasks.
- Level indicator.

A date picker is a control to choose the date and time. There are three date picker styles. The textual style uses a text field. The textual with stepper style uses a text field and steppers. The graphical style uses a calendar and clock.

A segmented control is a control that is divided into multiple segments. The back and forward buttons at the top of Finder windows demonstrate the use of segmented controls. The buttons are the two segments in the segmented control.

Image wells display an image in a frame. You can also use an image well as a target for the user to drag an image.

A path control represents a file path. It can be used to display a bread crumb trail. Xcode's jump bar uses a path control.

Level indicators indicate amounts graphically. A common use of level indicators is measuring the relevance of search results.

Formatters

Formatters control the display of data. The Formatters group contains the following formatters: number formatter, date formatter, custom formatter, and byte formatter. Drag a formatter to a text field or to a column in a table view or outline view. The formatter controls how numerical values or dates are displayed.

A custom formatter is an abstract formatter class. Create a subclass of `NSFormatter`, add a custom formatter to the xib file, and set the class to your `NSFormatter` subclass in the identity inspector. Use a custom formatter if you need to format text in a way that can't be handled with one of the other formatters.

Apple added the byte formatter in Xcode 4.4. A byte formatter controls the display of byte counts, such as the size of files.

Data Views

The Data Views group contains views used to display large amounts of data. It contains the following items:

- Table view
- Outline view
- Browser
- OpenGL view
- Collection view
- Collection view item
- Predicate editor
- Rule editor
- Source list

Table and outline views both display rows and columns of information. The main difference is an outline view can show hierarchical data. Xcode's project navigator is an example of an outline view. The list of messages in a mailbox in Mail is an example of a table view. The source list object is an outline view with the source list highlighting that Mac applications like iTunes, Mail, and the Finder use to display a sidebar. Read the "Source Lists" section later in this chapter for more information on source lists.

Browsers display lists of data. If the data is not hierarchical, the browser has one column. If the data is hierarchical, the browser displays a column for each level of the hierarchy. The column view in the Finder is an example of a browser.

OpenGL views display OpenGL content. If you're using OpenGL in your Cocoa application, use an OpenGL view.

Collection views are similar to table views, but a collection view's rows and columns are views, not cells. If you want something like iPhoto's list of events, where there is a photo for each event, you would use a collection view.

Predicate and rule editors are used to filter the information that appears in the other data views. If you've worked with rules in Mail, you've seen the predicate editor in action. The predicate editor opens when you create a new rule or edit an existing rule. A predicate editor is a specialized version of a rule editor and inherits from the rule editor's class, `NSRuleEditor`.

Cells

In addition to the views I covered last section, the Data Views group contains cells for table views and outline views. You can choose from the following cells:

- Text field cell
- Combo box cell
- Checkbox cell
- Pop-up button cell
- Segmented cell
- Slider cell
- Stepper cell
- Level indicator cell
- Image cell
- Custom cell
- Image and text table cell view
- Text table cell view

Table cell views require Mac OS X 10.7 or later. Table cell views inherit from `NSView` instead of `NSCell`. By using table cell views you can create complex tables in Mac applications like the tables iOS applications have. Mail in Mac OS X 10.7 is an example of an application that uses table cell views. Select a mailbox or folder. Mail uses table cell views to display the messages in the selected mailbox.

To add cells to your user interface, you must first drag a table view or an outline view to the window. Drag the cell to a column of the table view or outline view.

Layout Views

The Layout Views group contains container views, views that other views reside in. This group contains the following items:

- Custom view
- Vertical split view
- Horizontal split view
- Scroll view
- Tab view
- Box
- Vertical line
- Horizontal line

If you have special drawing needs, the custom view is for you. The custom view item is a placeholder for a view subclass that you create. Place the custom view in the window. Create a subclass of Cocoa's view class, `NSView`, and tell Interface Builder the custom view is the subclass you created.

A split view contains multiple views and lets the user adjust the size of each view inside the split view. The user can adjust the width of each view in a vertical split view. The user can adjust the height of each view in a horizontal split view. The "Split Views" section later in this chapter has additional information on creating split views.

A scroll view is a custom view that displays more information than can fit in the custom view. If your custom view is going to display a lot of information, use the scroll view instead of the custom view.

Tab views allow you to show multiple pages of information in a single window. Selecting a tab makes that tab's page appear in the window. Apple's System Preferences contain many examples of tab views.

A tab view initially has two tabs. To add more tabs, select the tab view and open its attributes inspector. Enter the number of tabs you want in the Tabs text field.

To place controls in a particular tab, select the tab. Drag the controls from the object library to the tab's view. Repeat for each tab.

Boxes divide a window into distinct areas. If your window has lots of user interface elements, using boxes makes your interface easier to comprehend. If you're going to use a box, start by creating the box. Drag the controls into the box. The vertical and horizontal line objects are special cases of boxes; they use lines to divide the window.

Objects and Controllers

The Objects and Controllers group contains elements that are not visible in your user interface. This group contains the following items:

- Object, which you use to add instances of your application's classes to the xib file so you can connect user interface elements to your classes.
- View controller, which manages a view.
- Object controller, which manages one object.
- Array controller, which manages a group of objects. Array controllers are the most widely used controllers.
- Tree controller, which manages hierarchical data.

- User defaults controller, which works with applications that save user preferences. The user defaults database stores the preferences that differ from the ones that shipped with the application.
- Dictionary controller, which manages a dictionary of key-value pairs.
- Managed object context, which manages a collection of Core Data managed objects.
- Popover and view controller.
- Text finder.
- Page controller.

Xcode's pop-up editors are examples of popovers. Select an element from the object library to see an example of a popover. A text finder is a controller for a find bar. You can see an example of a find bar in Safari by choosing Edit > Find > Find. Xcode's search navigator provides another example of a find bar.

Xcode 4.4 adds the page controller. Page controllers control page view animations. If your application displays multiple pages of information or uses swipes to switch pages, use a page controller.

Windows and Menus

The Windows and Menus group contains elements that apply to the application as a whole. This group has three unofficial categories of items: Windows, Menus, and Toolbar, which contains a toolbar and toolbar items.

Windows

The Windows group contains the windows you can add to your interface. To add a window, drag it from the object library to the canvas. You can add the following types of windows: window, panel, textured window, HUD window, and window and drawer. Add a window when you want to add a basic window, a dialog box, or a sheet to your application.

Panels are auxiliary windows that float above other windows. Examples of panels are Apple's color picker and font panel, which many Mac applications use to select the current text color and font.

A textured window is a window where the entire background's color matches the background of a normal window's title bar. Launch OpenGL Profiler to see a textured window. The window where you choose an application to profile is an example of a textured window.

A HUD window is a special kind of panel. You can see an example of a HUD window in Interface Builder by selecting an item in the canvas or object list and right-clicking. Right-clicking opens a HUD window that shows the selected item's connections.

A window and drawer includes a window, a drawer that is attached to the window, and a view for the drawer's content. The user determines whether or not the drawer is open. Drawers were used often in early versions of Mac OS X, but are not used as much now.

Menus

The Menus group contains the menus and menu items you can add to your application's menu bar. The group contains the following items:

- Menu
- Menu item
- Submenu
- Menu separator
- Application menu
- File menu
- Edit menu
- Find menu
- Text menu
- Format menu
- Font menu
- Window menu
- Help menu
- Full screen menu item

The menu bar for a Cocoa application contains the Application, File, Edit, Format, View, Window, and Help menus. Use one of the named menus to add a menu to the menu bar, even if your application needs a menu that's not in the list. Change the menu title to the title you want. Refer to the "Working with Menus" section later in the chapter for more information on working with menus.

Submenus work with hierarchical menus. Choose File > Open Recent in Xcode to see an example of a hierarchical menu. The Open Recent submenu contains the most recent projects you opened in Xcode. Menu separators group related items in a menu. If you have a lot of items in a menu, use menu separators to make finding an item in the menu easier.

Dock menus and contextual menus use the Menu object. A Dock menu opens when you right-click a Dock icon. Use the Menu object to create a custom Dock menu for your application. Contextual menus work with user interface elements. The menu opens when the user right-clicks the element. The contextual menu contains common commands for the user interface element.

The full screen menu item is a single menu item with the title Enter Full Screen. Most applications that use the full screen menu item place it in the View menu.

Toolbar

The Toolbar group contains the elements you need to create toolbars in Interface Builder. This group contains the following items:

- Toolbar
- Image toolbar item
- Separator toolbar item
- Flexible space toolbar item
- Space toolbar item
- Show colors toolbar item
- Customize toolbar item
- Show fonts toolbar item
- Print toolbar item

Out of the available toolbar items, the image toolbar item is the one you will use most because the image toolbar item is what you use to create custom toolbar items. For more information on creating toolbars, refer to the “Toolbars” section later in this chapter.

Address Book

The Address Book group has two objects: address book people picker view and address book person view. The people picker view lets your application work with the user’s address book data. Mail demonstrates the use of the address book people picker view. Choosing Window > Address Panel in Mail opens the address panel. Selecting a name from the address panel and clicking the To button addresses an email to that person.

The address book person view displays a person’s address book entry. Use the person view to examine and edit a person’s contact information.

Automator

The elements in the Automator group work with Automator action projects and application projects that support Automator workflows. The Automator group contains the following items:

- Automator path pop up button, which lets users choose a path in an Automator action.
- Workflow controller, which manages an Automator workflow in your application.
- Workflow view, which lets users view and edit an Automator workflow in your application.
- Automator token field, which is a token field (text field) that has support for Automator variables.

Disc Recorder

The Disc Recorder group has one item: a Minutes/Seconds/Frames formatter (MSF formatter). The MSF formatter calculates the length of CD and DVD tracks. CDs and DVDs store their data in frames, where 75 frames equal one second. The MSF formatter displays the frames in the human-readable form of minutes and seconds.

You cannot drag the MSF formatter to the window; the MSF formatter works with controls that display text to format the text. Controls that work with formatters include text fields, forms, search fields, combo boxes, and tables. First, place a control that works with formatters, such as a text field, in the window. Then drag the MSF formatter to the control. For tables, drag the formatter to the table column that should display the formatted text. If a control cannot handle a formatter, Interface Builder will reject your attempt to drag a formatter to the control.

If you have multiple controls that use an MSF formatter, drag the formatter to the canvas. Multiple controls can then access the formatter.

Image Kit

The Image Kit group contains views for displaying images and importing images from scanners and digital cameras. This group contains the following views:

- Scanner view, which allows you to scan an image.
- Camera view, which displays the contents of a digital camera.
- Device browser view, which lets you select a scanner or camera from a list of devices.
- Image view, which displays images.
- Image browser view, which you use to browse large numbers of images.

OSAKit

The OSAKit group is for Cocoa applications written in AppleScript. This group has the following items:

- Dictionary view for viewing the AppleScript terms an application supports.
- Dictionary controller for managing dictionaries.
- Script view for viewing and editing scripts.
- Script controller for managing scripts.

PDFKit

The PDFKit group is for applications that work with PDF files. This group contains two views: a PDF view to display PDF data and a PDF thumbnail view to display thumbnails of PDF data.

QTKit

QTKit is the framework that lets you access QuickTime from Cocoa applications. For those of you who don't know what QuickTime is, it is Apple's multimedia technology. With QuickTime you can record and play video. If your application works with video, the QTKit group is for you.

The QTKit group has two views. The first view is a movie view, which lets you play video in your application. The second view is a QuickTime capture view. A capture view stores a preview of the video your application captures.

Quartz Composer

The Quartz Composer group has elements that work with the Quartz Composer application. This group contains four items.

- Quartz Composer view
- Quartz composition picker view
- Quartz composition parameter view
- Quartz Composer patch controller

Use the Quartz Composer view to play a Quartz Composer composition in a Cocoa application. Use the Quartz composition picker view to browse and preview compositions. Use the Quartz composition parameter view to display the parameters of a composition. Use the Quartz Composer patch controller to create bindings between compositions and user interface elements.

SceneKit

Xcode 4.4 adds the SceneKit group, which contains one item: a scene view. A scene view displays a 3D scene using Apple's Scene Kit framework for 3D graphics. If you are writing a Cocoa application that uses Scene Kit, you should add a scene view to your window.

WebKit

The WebKit group contains one object: a web view. Use a web view to embed content from the World Wide Web in your application.

Custom Objects

The Custom Objects section contains third-party Interface Builder elements. Because Xcode 4 does not allow you to edit xib files that use third-party plug-ins, the Custom Objects section will usually be empty.

Media Library

The media library contains audio, image, and video files. Choose View > Utilities > Show Media Library to display the media library. The media library has two sections: System and Workspace. The System section contains system sounds and images. The Workspace section contains media files you have added to the projects in your workspace. If you haven't added media files to your projects, the Workspace section will be empty.

Selecting a file from the media library opens a pop-up editor for the file. You can play audio and video files from the pop-up editor and view image files.

Inspectors

Interface Builder has eight inspectors for Cocoa applications. The inspector area has a selector bar at the top with a button for each inspector. The inspectors, running from left to right in the selector bar, are the following:

- File
- Quick Help
- Identity
- Attributes
- Size
- Connections
- Bindings
- View effects

Some elements do not have all the inspectors. Controllers have empty view effects and size inspectors because they are not visible in an application.

The file inspector applies to the xib file as opposed to a specific user interface element. The Quick Help inspector is rarely used in xib files.

File Inspector

The file inspector has one section specifically for xib files: the Interface Builder Document section. There are three things you can set in the Interface Builder Document section of the file inspector: deployment target, development target, and localization locking.

Use the Deployment menu to choose the deployment target, which is the earliest version of Mac OS X that can load the xib file. If your interface has any problems on older versions of Mac OS X, they appear in the issue navigator. Choose View > Navigators > Show Issue Navigator to open the issue navigator.

Use the Development menu to choose the development target, which is the earliest version of Interface Builder that can open the document. Setting the development target can help when working with other developers who are using older versions of Xcode and Interface Builder.

Use the Localization Locking Default pop-up menu to choose what parts of the xib file should be locked. Initially nothing is locked, which is good when you're starting to build your user interface. When you have the interface finalized, you may want to lock the document so you don't accidentally change something. If you're going to translate your application to other spoken languages, you would lock the non-localizable properties so translators can translate your application's menus, titles, and labels without messing up the rest of your interface.

Quick Help Inspector

Each element in the object library has a corresponding class. For example, a window is an instance of the `NSWindow` class. Selecting an element from a window or from the object list provides information on the element's class in the Quick Help inspector as well as links to documentation on the class.

Identity Inspector

The identity inspector has the following sections:

- Custom Class
- Identity
- Tool Tip
- Accessibility Identity
- User Defined Runtime Attributes
- Document

Not every element has all the sections. Windows don't have the Accessibility Identity and Tool Tip sections.

Custom Class

Each user interface element in Interface Builder has a corresponding class. The Class combo box lets you set the class for an element. You must set the class if you create a subclass of an existing class and want to use that subclass in Interface Builder.

Identity

The Identity section contains an Identifier text field. This text field allows you to set a unique identifier for the user interface element. You do not need to enter an identifier. A unique identifier will be generated when the xib file loads. Set an identifier if you want to control the identifier's value.

Earlier versions of Xcode place the contents of the Document section in the Identity section.

Tool Tip

A tool tip is a short help message that appears when the user places the mouse cursor over a user interface element. If you want to display a tool tip, enter the message in the Tool Tip text field.

Accessibility Identity

The Accessibility Identity section lets you set attributes to make your program accessible to people with disabilities. The Description text field describes the element. Suppose your program has back and forward buttons that display arrows instead of text. By giving the buttons descriptions, a blind person would be able to understand what your buttons do.

The Help text field lets you enter a help message for people with disabilities. Enter the message in the text field.

User Defined Runtime Attributes

The User Defined Runtime Attributes section lets you create a dictionary of key-value pairs for a custom object that does not have its own Interface Builder inspector. The dictionary is added to the object when your application launches.

Click the + button to add a key-value pair to the dictionary. Each key-value pair has three elements.

- Key Path, which is the key name.
- Type, which is the data type.
- Value, which is the key's initial value.

A key can have one of the following data types: Boolean, Number, String, Localized String, and Nil. The data type determines what you can enter for the value. The Nil data type doesn't let you enter anything. The Boolean data type gives you a checkbox. The other data types use a text field. Double-click the value to enter the initial value for the key.

Xcode 4.4 adds the following data types for keys: Point, Size, Rect, Range, and Color. The Color data type has a color picker to choose the initial color. The other new data types use a text field to set the value.

Document

The Document section makes things easier for you in Interface Builder. You can give an object a more descriptive name, a color label, and add notes to make the object easier to identify in the editor and jump bar. When you name an element using the Label text field, the element's name also changes in the object list.

You can also lock an element so other people can't make changes to it. If you're going to translate your application to other spoken languages, you would lock the non-localizable properties so translators can translate the object's text without messing up the rest of your interface.

Earlier versions of Xcode place the contents of the Document section in the Identity section.

Attributes Inspector

The attributes inspector is where you set an item's basic attributes, anything not covered by the other inspectors. Use the attributes inspector to perform the following tasks:

- Give menu items and controls keyboard equivalents.
- Provide titles for windows and controls.
- Set the number of rows and columns in a radio button group.
- Set the initial value of radio button groups and checkboxes.
- Set a slider's minimum, maximum, and initial values.
- Set the number of tabs in a tab view.
- Set the number of columns in a table view.
- Give a text field placeholder text to display initially.
- Determine the controls a window should have: close button, minimize button, zoom button, and resize control.
- Set the background color of a view.
- Determine the information a date picker should show.
- Set the initial path for a path control.

Each user interface element has its own set of attributes, which means what you can set in the attributes inspector depends on the item.

Size Inspector

The size inspector controls the size, position, and alignment of user interface elements.

Sizing Controls

If you select a control, the size inspector has a Control section. Use the Size pop-up menu to set the size for a control. The size is initially Regular, but you can change it to Small or Mini. If a control has only one size, choosing Small or Mini has no effect.

Setting an Element's Size and Position

The View section of the size inspector has text fields to set an element's size and position. To reposition a view enter new values in the X and Y text fields. To resize a view enter new values in the Width and Height text fields.

Next to the text fields is the Origin field, which consists of a white square with nine dots. Clicking a dot tells you the dot's position. If you click the dot in the upper left corner, the X and Y text fields display the coordinates of the upper left corner of the element.

Autosizing

The autosizing square determines a view's position and size when the user resizes the window. This square has one square inside a larger square, which you can see in Figure 3.5. A preview window appears next to the square. The preview shows you what happens to the view when the window changes size.

If you enable auto layout for the xib file, the size inspector has no autosizing square. Auto layout replaces the autosizing square. Read the “Auto Layout” section later in this chapter to learn more about auto layout.

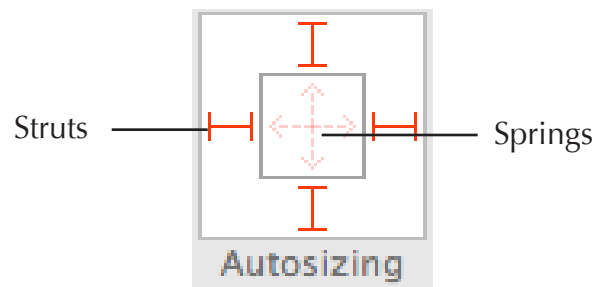


Figure 3.5

Autosizing square

Springs

The inner square has two springs: one for vertical resizing and one for horizontal resizing. Clicking a spring toggles it. If a spring is not highlighted, the object maintains its size and position in the window. Controls like buttons and checkboxes are items you don't normally resize. You usually don't want your buttons to get huge when the user makes the window bigger.

If a spring is highlighted (darker red), the object changes size when you resize the window. Views are the most common items to resize. If you're writing a text editor and the user makes the window larger, the text view should get larger as well.

Struts

The outer square has four struts: one for each side of the view's rectangle. Clicking a strut toggles it. If a strut is not highlighted, the object maintains its relative position in the window. If a view is in the center of the window, it remains in the center of the window when the user resizes the window.

If a strut is highlighted (darker red), the object keeps its literal position in the window. If the lower left corner of a view is at the position (150, 100) in the window, it stays at (150, 100) when the window resizes.

Aligning Elements

Below the View section of the size inspector is the Arrange menu. This menu allows you to align multiple elements. Select the elements you want to align and make a choice from the Arrange menu. The menu has sections to align the elements vertically and horizontally. You have the following vertical alignment options:

- Align top edges
- Align vertical centers
- Align baselines
- Align bottom edges

You have the following horizontal alignment options:

- Align left edges
- Align horizontal centers
- Align right edges

Choosing Editor > Align accomplishes the same thing as using the Arrange menu.

Positioning Items in a Containing View

The Arrange menu also allows you to position items in their containing view. Select the element or elements you want to position and make a choice from the Arrange menu. You can center horizontally or vertically in the containing view. You can also fill the containing view horizontally or vertically. Filling the containing view is normally done for larger views like custom views, OpenGL views, and text views. Filling vertically is a good way to ensure a source list (table or outline view) fills the window.

Sizing Windows

The size inspector works differently for windows. You can set a window's size, minimum size, maximum size, initial position, and content border.

The size inspector has a view that lets you set the initial position of the window. Refer to Figure 3.6. Drag the gray rectangle to position the window.

On the outside of the window are four struts. The struts are initially highlighted, which means the window is positioned proportionally both vertically and horizontally. Clicking the left strut makes the window position fixed from the right. Clicking both the left and right struts centers the window horizontally. Clicking the other two struts works similarly to clicking the left and right struts. You can also use the pop-up menus below the struts to do the positioning instead of using the struts.

For a content border, you can add a small or large border at the bottom of the window, have an autosized border, or create your own custom borders at the top and bottom of the window. If you choose something other than an autosized content border, it won't run on anything prior to Mac OS X 10.6.

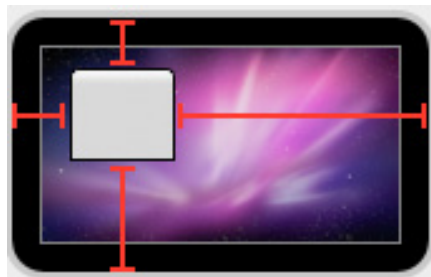


Figure 3.6

Setting a window's initial position

Connections Inspector

Cocoa applications use messages to send data from one object to another. Connections let your user interface elements send messages to each other, send messages to models and controllers, and receive messages from models and controllers. The connections inspector is where you manage your user interface elements' connections to other objects. The connections inspector shows an element's outlets, accessibility outlets, actions, and bindings.

Outlets are variables in a class that can connect to other classes or controls. The connections inspector has two sections for outlets: Outlets and Referencing Outlets. The Outlets section shows you the outlets that can be set for the selected element. The Referencing Outlets section lists any outlets in other elements that reference the selected element.

Accessibility outlets are outlets that help people with disabilities use your application's user interface. Like outlets, the connections inspector has two sections for accessibility outlets: Accessibility and Accessibility References.

Actions are methods in the selected element's Cocoa class. Other classes send a message to the element telling it to perform the action. The connections inspector has two sections for actions: Sent Actions and Received Actions. The Sent Actions section lists any actions the selected element sent to other elements. It lists the connections where the selected element is the sender. The Received Actions section shows you the possible actions the selected element can receive, actions other elements send to this element.

The Bindings section lists any bindings the selected item initiated, such as binding a controller to File's Owner. The Referencing Bindings section lists any bindings where another element bound itself to the selected item, such as binding a view to a controller.

If there is a connection, the outlet, action, or binding is highlighted with a destination, the connecting object, appearing next to it. The radio button next to the outlet, action, or binding is selected as well, which you can see in Figure 3.4. To remove a connection, click the close button (it's an X) for the connection in the inspector.

Bindings Inspector

In the Model-View-Controller design pattern, programmers spend a lot of time writing code for controller classes that supply data from model classes to view classes and vice versa. Cocoa bindings reduce the amount of code you have to write. Cocoa has controller classes, which you can find in the Objects and Controllers section of the object library, that save you from writing code. Bindings connect user interface elements to these controller classes.

The bindings inspector is where you view the bindings each user interface element has and bind the elements to controllers. To create a binding, select it from the bindings inspector and select the Bind to checkbox. The section “Bindings” later in this chapter has detailed instructions on how to create a binding.

View Effects Inspector

The view effects inspector is used with Core Animation, Apple’s technology for creating animated user interfaces. Core Animation is a large topic. I know of two books that have been written on Core Animation so there’s no way I can adequately cover Core Animation in this section. Read the *Core Animation Programming Guide*, which is part of Apple’s documentation, for more information on Core Animation.

Turning on Core Animation Effects

A list of views appears at the top of the view effects inspector. This list represents the view hierarchy. The view you selected is at the bottom of the hierarchy, and the window’s content view is at the top of the hierarchy. Each view in the hierarchy has a checkbox next to it. Selecting the checkbox adds Core Animation effects to the view and any child views. If you select the checkbox for the window’s content view, Core Animation effects are applied to everything inside the content view.

Appearance Section

The Appearance section lets you set the alpha value and display a shadow. Alpha values range from 0 to 1, where 0 is transparent and 1 is opaque. If you enable shadows, you can specify an offset, radius, and color for the shadow.

Content Filters

Content filters are Core Image effects that are applied to a view and its subviews. Click the + button to add a content filter. When you add a content filter, the Filter pop-up menu is enabled. Choose a filter from the menu. Interface Builder has the following filter groups:

- Geometry Adjustment
- Distortion Effect
- Blur
- Sharpen
- Color Adjustment
- Color Effect

- Stylize
- Halftone Effect
- Tile Effect

What you can specify for a filter depends on the filter you choose. For the Crop filter, you specify the crop rectangle. Most of the blur effects let you specify a radius for the effect.

Background Filters

Background filters are Core Image effects that are applied to content behind the view. Click the + button to add a background filter. When you add a background filter, the Filter pop-up menu is enabled. Choose a filter from the menu. The background filters are the same as the content filters.

Compositing Filters

Compositing filters combine a layer's content with the layers behind it. Choose a compositing filter from the pop-up menu. Most of the available compositing filters are blend modes.

When you choose a compositing filter, the Background Image combo box appears. Use the Background Image combo box to add a background image to the compositing filter.

Transitions for Subviews

Use the Subviews pop-up menu to add a transition animation for a subview. There are two groups of transitions in the menu. At the top of the menu are the custom transitions: Fade, Move In, Push, and Reveal. Below the custom transitions are the built-in transitions: Bars Swipe Transition, Copy Machine, Disintegrate With Mask, Dissolve, Flash, Mod, Page Curl, Ripple, and Swipe.

After choosing a transition, configure it. What you can configure depends on the transition you choose. The custom transitions have progress controls that let you set the start and the end of the transition. The Move In and Push transitions have a pop-up menu to specify a subtype, which can be top, bottom, left, or right.

All of the built-in transitions let you specify a target image and a length of time. Use the Target Image combo box to specify the target image. Use the Time text field to specify a length of time, which can range from 0 to 1 second.

Working with Menus

Every Mac OS X GUI application has a menu bar, which means every application has to work with menus. Interface Builder can handle the most common menu-related tasks, including the following:

- Adding menus to the menu bar.
- Adding menu items to a menu.
- Adding submenus to a menu.
- Giving a menu item a keyboard equivalent.
- Creating contextual menus.
- Creating Dock menus.
- Attaching menus to buttons.

Adding Menus to the Menu Bar

Adding a menu to the menu bar is relatively easy. Select a menu from the object library and drag it to the menu bar. If the menu isn't in the right location, select it from the menu bar and drag the menu to the proper location.

Looking at the Xcode menu bar, you can see several menus, Navigate, Editor, and Product, that aren't in the object library. How do you add menus that aren't in the object library? Drag one of the menus from the object library to the menu bar, and double-click the menu bar to change the menu name to what you want. From there you can add, remove, and modify the menu items in your new menu to reflect your application's needs.

Adding Items to a Menu

Adding items to a menu is simple. Select the Menu Item element in the object library. Drag the Menu Item element to the menu in the menu bar where you want to place the menu item. Dragging the item to the menu opens the menu and shows you the items in the menu. Drag the item to where you want it to appear in the menu. You may find it easier to open the menu before adding the item to it. Click the menu in the menu bar to open the menu. If you're adding an item to a pop-up menu, double-click the menu to open it before adding the item.

The menu item you added has the name Item, which is most likely not the name you want. Double-click the item to change the menu item's text. You can also change the names of existing menu items. Remember from the previous section that you can drag a menu from the object library and change its name to make it your own menu. If you change the menu's name, you'll want to change the menu items as well.

To delete a menu item, select it and press the Delete key. Deleting menus from the menu bar works the same way. Select the menu from the menu bar and press the Delete key to remove the menu from the menu bar along with all the menu's items.

Keyboard Equivalents

Many standard Mac menu items have keyboard equivalents. Pressing Command-Q is the equivalent of choosing AppName > Quit. To give a menu item a keyboard equivalent, select the menu item. The item's name is on the left and the keyboard equivalent is on the right. The keyboard equivalent is initially blank. Double-click the right side of the menu item to set the keyboard equivalent. Enter the keyboard equivalent.

If you make a mistake setting a keyboard equivalent, select the menu item and open the attributes inspector. Click the Clear button on the right side of the Key Equivalent text field to remove the keyboard equivalent. Enter a new keyboard equivalent.

Adding Submenus

A special menu item is a submenu, a menu item that contains additional menu items. You can see an example of a submenu in Xcode's File menu. The menu item Open Recent is a submenu; the additional items are the projects you opened most recently.

Adding a submenu is the same as adding an ordinary menu item. Drag the submenu to the menu. After adding the submenu to the menu, you can add items to the submenu. To add items to the submenu, select the submenu. The submenu's items will become visible. Drag a menu item to the list of items, and that menu item will become part of the submenu.

Creating Contextual Menus

Contextual menus work with user interface elements. When the user right-clicks the element, the contextual menu opens. The contextual menu contains frequently used commands. If you're writing a text editor and the user selects some text, a contextual menu for the text view would have items for cutting, copying, and pasting text. The point of contextual menus is to keep the user from having to move the mouse cursor to the menu bar to perform a menu command. To create a contextual menu, perform the following steps:

1. Drag a Menu object to the canvas.
2. Add the items you want to add to the menu.

3. Make a connection from the control to the Menu object. Select the control, hold down the right mouse button, and drag the mouse to the menu.
4. When you make the connection, the connections panel for the control opens. Select menu.

Creating Dock Menus

Every application in the Dock has a menu the user can display by right-clicking the application's icon. If the application is running, the Dock menu has items to quit and hide the application as well as items for each open window in the application. On Mac OS X 10.7 and later the Dock menu contains a list of recently opened documents. These items appear automatically for your application as well. Creating a Dock menu with custom menu items requires you to take the following steps:

1. Open the `MainMenu.xib` file. You can't add a Dock menu to the document's xib file in a document-based application.
2. Drag a Menu object to the canvas.
3. Add the items you want to add to the menu. Add only the custom items that aren't part of the normal Dock menu.
4. Make a connection from the File's Owner to the Menu object. Select File's Owner from the object list, hold down the right mouse button, and drag the mouse to the menu.
5. When you make the connection, the connections panel for the File's Owner object opens. Select `dockMenu`.

Attaching Menus to Buttons

Menus are not limited to the menu bar. You can attach them to buttons as well. The Mail application has a button with an attached menu in the lower left corner of the main window. Many Mac applications have icon buttons with menus in their toolbars. To attach a menu to a button, perform the following steps:

1. Drag a pop-up button to the window.
2. Open the pop-up button's attributes inspector.
3. Choose Pull Down from the Type pop-up menu.
4. Choose a button type from the Style pop-up menu.
5. If you have an icon button, use the Image combo box to pick an image for the button.
6. If you have an icon button, deselect the Bordered checkbox.

Bindings

Bindings allow your Cocoa application to use controller objects to manage relationships between model and view objects. They reduce the amount of code you have to write. There are four tasks to perform to create a binding.

- Create the model class.
- Create the controller.
- Bind the model to the controller.
- Bind the view to the controller.

Creating the Model Class

How you create the model class depends on whether or not you use Core Data. If you use Core Data, you will use Xcode's data modeling tools to create your models. If you don't use Core Data, you will have to create a model class in Xcode. Choose File > New > File to create a new file. In most cases you'll want a subclass of `NSObject`.

Creating the Controller

Creating the controller is the easiest step. Select a controller from the object library and drag it to the canvas. The object library has the following controllers:

- A view controller manages a view.
- An object controller manages one object.
- An array controller manages a group of objects.
- A tree controller manages hierarchical data. Tree controllers are used most often with outline views.
- A dictionary controller manages a dictionary of key-value pairs.
- A user defaults controller manages user preferences.
- A text finder is a controller for a find bar, which is used for search and replace.
- A page controller controls page view animations.

Binding the Model to the Controller

After creating the model and the controller, you must bind the model to the controller. Select the controller from the object list and open the attributes inspector. The attributes inspector for object controllers, array controllers, and tree controllers has an Object Controller section. You should see a pop-up menu called Mode. There are two options for mode: Entity and Class. Core Data applications should choose Entity. Other applications should choose Class.

If you have a Core Data application, type the name of the entity in the Entity Name text field. If you don't have a Core Data application, type the name of your model class in the Class Name text field.

Most controllers also require you to add keys to the controller. These keys should correspond to data members of your model class. For controllers that have an Object Controller section, click the + button to add a key. Dictionary controllers have a Dictionary Controller section in the attributes inspector where you can include and exclude keys from the dictionary.

View controllers and user defaults controllers have limited options for binding models to controllers.

Binding the View to the Controller

After binding the model to the controller, the final step is to bind the view to the controller.

1. Select the view and open the bindings inspector.
2. Select a binding from the list by clicking the disclosure triangle next to it. Each user interface element has its own list of bindings.
3. Select the Bind to checkbox to turn on the binding.
4. Choose a controller from the pop-up menu next to the Bind to checkbox.
5. Choose the controller key, which is a key in the controller class.
6. Choose the model key path, which is a key in the model class.

Selecting the Raises for not applicable keys checkbox tells Xcode to raise an exception in your application if the controller key or model key path does not apply for a piece of data.

Each binding allows you to set placeholders for multiple values, no selection, not applicable values, and null values. The placeholder has a text field or a pop-up menu, depending on the element and the binding. Enter a placeholder value in the text field or choose one from the menu.

Value Transformers

Value transformers are functions that change the value of a piece of data. If a binding uses a value transformer, the model's data goes through the value transformer before the user interface element displays the data. The Cocoa framework comes with five value transformers.

- `NSNegateBoolean`, which you should use to select checkboxes and radio buttons.
- `NSIsNil`, which you can use to enable and disable user interface elements.
- `NSIsNotNil`, which you can use to enable and disable user interface elements.
- `NSUnarchiveFromData`, which takes an `NSData` object and unarchives it. User defaults controllers use this transformer.
- `NSKeyedUnarchiveFromData`, which takes an `NSData` object that uses keyed archiving and unarchives it. Core Data uses this transformer to get attributes from `NSData` objects.

Subclass `NSValueTransformer` to create custom value transformers that perform data conversions. Suppose you store a piece of data in your program as a hexadecimal number, but want to display it as a decimal number. The custom value transformer makes the hexadecimal to decimal conversion.

The bindings inspector has a combo box to select a value transformer. The combo box's menu contains the available built-in value transformers for the view you're binding. If you want to use a custom value transformer, you must type the transformer's name in the combo box.

Connecting to Your Classes

One of Interface Builder's most powerful features is its ability to make connections from user interface elements to your application's classes. You can add outlets, actions, and bindings to your classes from Interface Builder. Adding connections to your classes requires the following steps:

1. Add an Objective-C class to your project.
2. Add an Object from the object library to the xib file.
3. Set the Object's class to your class.
4. Open the assistant editor.
5. Make connections to the class's header file.

If you haven't created your class, choose `File > New > File`. Select Cocoa under Mac OS X on the left side of the New File Assistant. Select Objective-C class. Click the Next button. Enter a class name and choose a subclass. Click the Next button. Pick a location to save the file and click the Save button.

Drag the Object item from the object library to the canvas to add an instance of your class to the xib file. Select the Object item from the object list and open the identity inspector. Choose your class from the Class combo box at the top of the inspector.

Opening the assistant editor enables you to have both the xib file and the header file visible so you can make connections to the header file. Click the center button in the Editor group in the project window toolbar to open the assistant editor.

To make a connection, right-click on an Interface Builder object (any UI element) and drag to the header file. The interface for a newly created Objective-C class looks something like the following code:

```
@interface MyClass : NSObject {

}

@end
```

The type of connection you can create depends on where you drag in the header file. If your drag destination is inside the braces of the `@interface` declaration, you can add an outlet to your class. If your drag destination is between the closing brace and the `@end` directive, you can add an outlet or an action. Sometimes you can't create an action, depending on the item you drag to the header file. Creating an outlet outside the closing brace creates a property for the outlet so you don't have to write accessor methods for the outlet. Creating an action also creates an empty method for the action in the implementation file. If the drag destination is a variable in the class, you can create a binding.

When you create the connection, a pop-up editor opens. For an outlet or action you can name the outlet or action and specify its type, which is the data type for an outlet and the return type for an action. If you have a choice between outlet and action, use the Connection pop-up cell to choose. Click the Connect button to finish creating the connection.

If you're creating a binding, use the Bind pop-up cell to choose the binding. Use the Controller pop-up cell to choose a controller. Enter the class of the controlled object and the key path for the controller. Enter a key path for the model. Click the Connect button to create the binding.

Grouping Objects

Grouping user interface elements can make working with them easier in Interface Builder and can make your user interface easier to navigate. Select the objects in the canvas that you want to group together. Choose Editor > Embed In. You can embed objects in a box, a custom view, a scroll view, a split view, a tab view, a matrix, or a submenu. If you want to unembed what you embedded, select the embedded view and choose Editor > Unembed.

Creating a Matrix of Controls

In Cocoa applications a matrix is a group of cells. A matrix lets you create a group of controls. Buttons and checkboxes are the controls you would most likely group in a matrix. Perform the following steps to create a matrix of controls:

1. Drag the control to the window.
2. Select the control.
3. Choose Editor > Embed In > Matrix.
4. Select the matrix and open its attributes inspector.
5. Use the Cells steppers to add rows and columns to the matrix.

When adding rows to the matrix, the newly added rows appear at the top of the matrix. If the matrix is near the top of the window, the newly added rows will not be completely visible. You must drag the matrix to see the new rows.

Setting Tab Order

If you have text fields in your window, pressing the Tab key lets the user move from field to field without taking her hands off the keyboard. When you lay out the interface, the initial tab order reflects the location of the controls in the window. Cocoa's initial tab order works well in many cases, but if you don't like the initial tab order, you will have to set the tab order yourself.

If you look through Interface Builder's inspectors, you'll see that setting the tab order is not obvious. How do you set the tab order for the controls in your window?

Use the `nextKeyView` outlet, which tells your program the control to move to when the user presses the Tab key. Set the `nextKeyView` outlet for a view to determine where your program should go when the Tab key is pressed. Make a connection from the view to the next view in the tab cycle.

You also must set the `initialFirstResponder` outlet to the first view in the tab cycle. Make a connection from the window to the view you want to start the cycle.

Toolbars

Toolbars are prevalent in Mac applications, especially Apple's applications. Open Xcode, Mail, Safari, or iTunes, and you will see a toolbar at the top of the window. You can add a toolbar to your application in Interface Builder without having to write any code. Creating a toolbar is a three-step process.

1. Add a toolbar to the window.
2. Add items to the toolbar.
3. Add images and labels to the toolbar items.

Adding a Toolbar

To add a toolbar to a window, drag a toolbar from the object library to the window.

Image and Custom View Toolbar Items

Interface Builder provides two kinds of toolbar items: image and custom view. Use a custom view toolbar item when you want an Interface Builder user interface element in the toolbar. The search field in Mail's toolbar is an example of a custom view toolbar item.

Use an image toolbar item when you want the toolbar item to behave like an icon button. Xcode's preferences window contains many examples of image toolbar items. The decision to use image or custom view toolbar items is not an either-or proposition. Your toolbar can contain both image and custom view toolbar items.

Adding Items to the Toolbar

Adding an item to the toolbar is a two-step process. The first step is to add the item to the list of allowed toolbar items. The list of allowed toolbar items contains the items the user can add to the toolbar when customizing the toolbar. Click the toolbar to open a sheet with a list of allowed toolbar items. Drag the toolbar item from the object library to the list of allowed toolbar items. The object library has an image toolbar item, which you should use to add image toolbar items. For custom view toolbar items, select the user interface element you

want to use from the object library and drag it to the list of allowed toolbar items. Suppose you want a search field in your toolbar. Drag the search field to the list of allowed toolbar items.

The second step is to make the item part of the toolbar's default toolbar item list, which shows how the toolbar looks the first time someone launches your application. If you want a toolbar item to be part of the default toolbar items list, drag the item from the list of allowed items to the default toolbar item list, which is just below the list of allowed toolbar items.

Interface Builder's initial toolbar configuration has the Colors, Font, and Print toolbar items. If you don't want those items in your toolbar initially, remove them from the default toolbar item list. If you don't want the Colors, Font, or Print toolbar items to ever be in your toolbar, remove them from the allowed toolbar items list.

Adding Images and Labels to Toolbar Items

After adding items to the toolbar, you must add a label for each item and add an image for each image toolbar item. Select a toolbar item from the jump bar or object list and open the attributes inspector. Use the Image Name combo box to pick an image file. Initially there are several Apple generic images for you to choose. If you have a custom image file you want to use for the toolbar item, add it to your Xcode project first.

Use the Label and Palette Label text fields to name your toolbar item. The Label text field specifies the label for the item in the toolbar. The Palette Label text field specifies the label for the item when customizing the toolbar. In most cases the Label and Palette Label values should be identical.

Split Views

A split view contains multiple views and lets the user adjust the size of each view inside the split view. In a vertical split view, the user can adjust the width of each view. In a horizontal split view, the user can adjust the height of each view. Split views are popular in Mac OS X applications. Xcode's project window provides examples of both horizontal and vertical split views. The project window has a vertical split view that contains the navigator, the editor, and the utility area. The utility area is an example of a horizontal split view. It contains the inspectors and the libraries.

Interface Builder provides both horizontal and vertical split views. To add a split view to a window, drag the split view from the object library. When you drag a split view to a window, two custom views are created for you.

Adding and Removing Views

To add a view to the split view, drag it from the object library to the split view. Make sure you select the split view to avoid making the new view a subview of one of the views inside the split view. To remove a view from the split view, select the view and press the Delete key.

Arranging and Sizing Subviews

The easiest way to arrange a split view's subviews is from the object list. Make sure the object list is in the hierarchical view so you can see the name of the split view. Click the disclosure triangle next to the split view to show its subviews. Select a subview and drag it to where you want it to be in the split view.

Use the dividers to size the split view's subviews.

Embedding a Split View

You can create a split view without dragging one from the object library. Drag two views to the window if you haven't already done so. Select them and choose Editor > Embed In > Split View. You may find embedding views easier than creating a split view when using Interface Builder's built-in views, like table views, text views, and outline views. Embedding views gives you the freedom to arrange and size the subviews the way you want before creating the split view.

An embedded split view is initially a vertical split view. If you want a horizontal split view, open the split view's attributes inspector and choose Top to Bottom from the Arrange menu. If your version of Xcode does not have an Arrange menu, deselect the Vertical Layout checkbox.

Dividers

Use the Style menu in the attributes inspector to set the type of divider to use for the split view. You can have a thick divider, a thin divider, or a pane splitter. Xcode's split views use the thin divider. The thick divider and pane splitter have a dimple in the center that lets the user know the splitter bar is draggable. The pane splitter has a 3D appearance.

Source Lists

What is a source list? It's a part of a window that lets you navigate and select objects in an application. Apple's applications provide many examples of source lists. You can see source lists on the left side of windows in the Finder, iTunes, and iPhoto. Xcode's navigator is also an example of a source list. Interface Builder makes it easy to add a source list to your application. To create a source list, perform the following steps:

1. Drag a table view or Source List object to the window. You can skip the rest of the steps if you use the Source List object.
2. Select the table view.
3. Open the attributes inspector.
4. Choose Source List from the Highlight pop-up menu.

In many applications source lists are part of a split view. If you want to use a split view, drag another view to the window. Select both views and choose Editor > Embed In > Split View.

Auto Layout

Auto layout maintains relationships between views. Auto layout can keep views from getting too large, too small, too close to other views, or too close to the edges of the window. Suppose you have two buttons next to each other in a window. You change the text of the left button so the text is twice as long. Before auto layout you would have to make the following changes manually: resize the left button to fit the new text and move the right button to maintain its position with the resized left button. With auto layout the left button resizes automatically to reflect the new text, and the right button moves to the proper position. Auto layout makes localizing user interfaces much easier.

Apple added auto layout in Mac OS X 10.7. Auto layout does not work on earlier versions of Mac OS X. If you use auto layout the xib file will not load in earlier versions of Mac OS X.

Turning on Auto Layout

If you create a new Cocoa application project, you don't have to worry about turning on auto layout. But if you have an existing project that you want to convert to auto layout, you must enable it. To enable auto layout, open the file inspector for the xib file. Select the Use Auto Layout checkbox in the Interface Builder Document section of the file inspector. An alert opens asking if you want to enable auto layout and change the xib file's deployment target to Mac OS X 10.7. Click the Continue and Upgrade button to finish.

When you turn on auto layout, the Autosizing section of the size inspector is removed. Auto layout replaces the autosizing mask.

Constraints

When you enable auto layout you won't notice any difference in the xib file until you select something inside the window. Selecting a control shows the control's constraints, which look similar to the struts in the size inspector's Autosizing section. Constraints are objects that express relationships between views.

Figure 3.7 shows an example of the constraints when you select a button. There are four constraints in the example: two for the Test button and two that involve both buttons. The constraints for the Test button keep the button from getting too close to the left edge and the bottom of the window. The constraints that involve both buttons keep them from getting too close to each other and keep their baselines aligned.

Editing Constraints

To modify a constraint open the attributes inspector and select the constraint. You may find it easier to use the jump bar to select a constraint. Click the View item in the jump bar and choose Constraints to open a menu of constraints. What you can control depends on the constraint, but most constraints have three fields: a Relation pop-up menu, a Constant text field, and a Priority slider.

The Relation menu has three possible values: Equal, Less Than or Equal, and Greater Than or Equal. The horizontal space between two controls is an example of a constraint that would have an equal relation. A less than or equal relation keeps views from getting too large or getting too far away from another view or the edge of the window. A greater than or equal relation is typically used with the height and width of large views like text views. They ensure the view doesn't get so small that it becomes unusable.

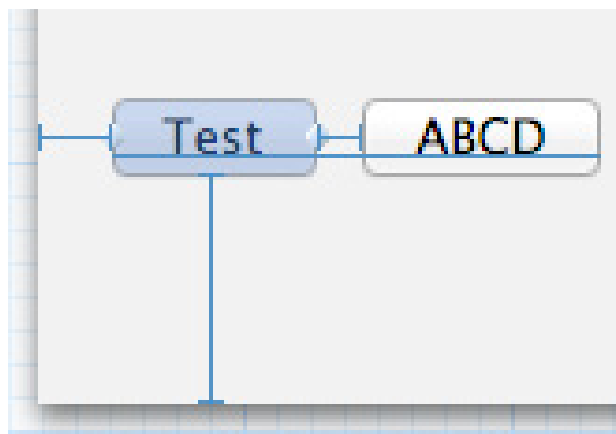


Figure 3.7

Constraints for a selected button

The Constant text field stores the value the Relation menu uses. Suppose you have a text view and you want it to always be at least 200 pixels wide. You would set the relation to greater than or equal and enter 200 in the Constant field.

Some constraints, usually ones that enforce space constraints, have a Standard checkbox next to the Constant text field. Selecting the Standard checkbox tells Interface Builder to use Apple's standards, which keeps you from having to know how much space there should be between two buttons or having to know how close to the edge a control should be. When you select the Standard checkbox, the Constant text field says Auto.

The Priority slider measures when the constraint should be enforced. The value of the priority can range from 0 to 1000. 1000 means always enforce the constraint. 0 means every other constraint or action, such as the user resizing the window, takes precedence over the constraint. When you drag the slider, a popover opens that explains what the priority means in practice.

Content Priorities

If you select a user interface element and open the size inspector, you will see two sets of sliders: content hugging priority and content compression resistance priority. Each set has two sliders: horizontal and vertical. The value for each slider can range from 0 to 1000, with 1000 being the highest priority.

The content hugging priority measures how much the edges of the element hug the element's content. The higher the content hugging priority, the less likely the element will resize. A button normally has high vertical content hugging priority and lower horizontal content hugging priority. Horizontal padding on a button is more acceptable than vertical padding.

The content compression resistance priority measures how likely the element's content is to be clipped or compressed. The higher the priority, the less likely the element's content will be clipped. A button should have higher content compression resistance priority than a text view. Clipping a button's title can make it unusable while clipping a text view's content allows the user to read at least some of the text in the view.

Adding Constraints

When you place user interface elements in a window, Xcode automatically creates constraints based on where you place the items. But you can also create your own constraints by using the Editor menu. Choosing Editor > Pin is the most common way of adding a constraint to a single view. Choosing Editor > Align is the most common way of adding constraints to multiple views.

Xcode 4.5 adds a set of three buttons to the lower right corner of the canvas to make working with auto layout easier. The left button aligns and centers views. The center button pins a view's size and spacing. The right button lets you determine how constraints are applied when resizing views. You can apply constraints to the resized view's descendants or to its siblings and ancestors.

Constraints you create and constraints you edit in the attributes inspector have thicker lines than the constraints Interface Builder automatically creates.

Chapter 4

Creating User Interfaces for iOS Applications

User interface separates iOS applications from other mobile applications. Cocoa Touch developers use Interface Builder to build their user interfaces. Xcode 4 integrates Interface Builder with Xcode, which allows you to build your user interface without leaving Xcode. This chapter shows you how to use Interface Builder to build user interfaces for iOS applications.

If you read the previous chapter, you're going to find I repeat some things in this chapter. I want this chapter to be a standalone resource for iOS developers so they don't have to go back and read the Mac chapter. Interface Builder has enough differences when creating Mac and iOS interfaces to warrant separate chapters for Mac and iOS.

Starting with Interface Builder

When you create an iOS application project, Xcode includes one or more xib files. Each xib file contains a window for you to place user interface elements. Select a xib file from the project navigator to open it. If you can't find your xib file, enter `xib` in the search field at the bottom of the project navigator to show all the xib files in your project.

When you select a xib file from the project navigator, the editor shows the user interface area. There are three sections that make up the user interface area, which you can see in Figure 4.1. On the left is the object list. Click the small button in the lower left corner of the canvas to toggle the icon and hierarchical views of the object list. Next to the object list is the canvas, which is where you lay out the visual elements of your interface. On the right is the utility area. If the utility area isn't open, open it by clicking the right button in the View group on the right side of the project window toolbar. The utility area contains the user interface elements and inspectors. The elements are in the object library at the bottom of the utility area. If you don't see the object library, choose View > Utilities > Show Object Library.

Above the canvas is the jump bar. The jump bar allows you to quickly select an object in the xib file. The jump bar helps you select a user interface element that is nested in a deep hierarchy.

Creating the User Interface

Before you create your interface, make sure Interface Builder's utility area is open and the object library is visible. The utility area contains the object library, which contains the elements you use to build the user interface. Choose View > Utilities > Show Object Library to show the object library, which you can see in Figure 4.2, at the bottom of the utility area.

To lay out the interface, select an element from the object library and drag it. Where you drag the element depends on the element. Elements that will be visible to users of your application, such as views, text fields, and buttons, should be dragged to the window. Controllers and windows should be dragged to the canvas.

When placing user interface elements in the window, Interface Builder provides guide lines to make sure elements line up with each other and to keep elements from being too close to the edges of the window. If you drag a view to an empty window and leave it over the window, the view expands to cover the entire window. This behavior helps when you want a view to cover the whole window.

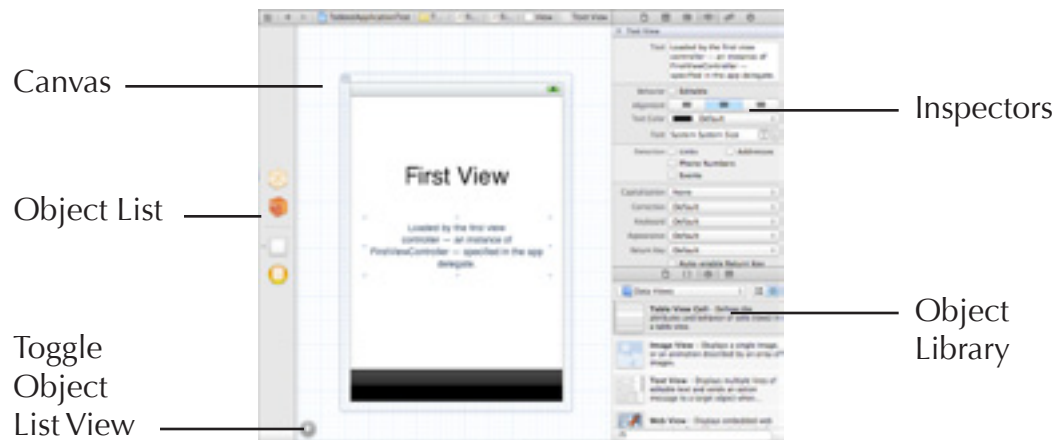


Figure 4.1

User interface area

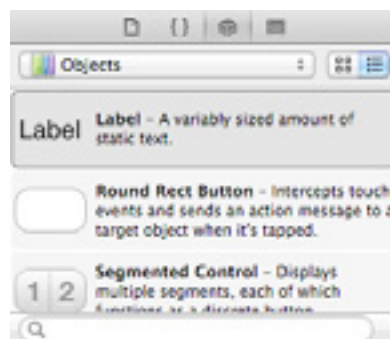


Figure 4.2

Object library

Modifying the Interface

When you lay out your program's user interface, the interface isn't going to look right initially. Buttons have no text, and labels display placeholder text. The size of your views may not be correct. In this section you'll learn how to make changes to your interface.

Selecting an Element

Before you can move, resize, or remove an element, you must first select it. Click on an item in the window to select it. If you want to select multiple elements, command-click the additional elements.

Selecting an Element in a Hierarchy

Because the iPhone and iPad have a smaller screen than a computer monitor, iOS applications are less likely to have deep hierarchies, where one element obscures another. But if you find one of your elements is obscured by another element, use the jump bar, which is above the canvas. The jump bar shows the hierarchy of items in the user interface. Click an item from the jump bar and use the submenus to get to the element you want to select.

Moving and Resizing Elements

Dragging the selected item changes its location in the window. To resize a control or view, click one of the eight dots bordering the selected control, hold down the mouse button, and move the mouse. Some iOS controls, such as text fields, have a fixed height. Controls with a fixed height have two dots instead of eight.

Deleting an Element

Select the element and press the Delete key to delete it.

Changing the Text of Titles and Labels

Changing the text of buttons and labels is easy in Interface Builder. Double-click the text you want to change and enter the new text.

Making Other Modifications

Use the inspectors at the top of the utility area.

Making Connections

Cocoa Touch programs use messages to send data from one object to another. Connections let your user interface elements send messages to each other, send messages to model and controller objects, and receive messages from model and controller objects. One of Interface Builder's most powerful features is the ability to make connections between objects without having to write code.

A connection involves two objects: the sender and the receiver. To make a connection, right-click the sender. Drag to the receiver. A HUD window opens with a list of outlets and events the receiver supports. Select an outlet or event from the HUD window.

To remove a connection you previously made, open the connections inspector by choosing View > Utilities > Show Connections Inspector. The connections inspector shows the connections an object has. Figure 4.3 shows an example of a connection. Select the connection and click the X button to remove it.

Testing the User Interface

Interface Builder lets you test your interface without having to build your Xcode project. Choose Editor > Simulate Document to launch the iOS Simulator, which is where you test the interface. Quit the simulator to stop testing and go back to editing your interface.



Figure 4.3

A connection in the connections inspector

Creating a Xib File

Add a new xib file to your Xcode project by choosing File > New > File or by dragging a xib file from the file template library to the project navigator. If you choose File > New > File, you can find the xib files for iOS applications in the User Interface section under iOS.

iOS applications have four xib files to choose from: application, empty, view, and window. Because Xcode's iOS application project templates include a xib file with a window, you normally don't need to create an application xib file, but it is available if you want to create a test interface.

NOTE

The xib files for iOS in the file template library are sized for the iPhone screen. To create a xib file for the iPad, choose File > New > File.

Object List

To the left of the canvas is the object list. As the name suggests, the object list contains a list of the xib file's objects. The object list has two views: icon view and hierarchical view. The icon view has icons for each object. The hierarchical view lets you see the names of the objects. The hierarchical view is easier to read, but the icon view takes up less space. Click the small button in the lower left corner of the canvas to change views.

The object list groups the objects into placeholders and objects. The hierarchical view has separate sections for placeholders and objects. The icon view uses a line to separate the placeholders and objects. The most common placeholders are File's Owner and First Responder. The Objects section contains the objects you add to the xib file, such as windows and controllers.

File's Owner

The File's Owner is the object that owns the xib file. The object doesn't exist until your application launches so you have no control over the object itself, but you can specify the class of the File's Owner.

1. Select the File's Owner object from the object list.
2. Open the Identity inspector by choosing View > Utilities > Show Identity Inspector.
3. Choose a class from the Class combo box.

For the xib files Xcode creates for iOS application projects, the default File's Owner is the `UIApplication` class.

First Responder

The First Responder object represents the first responder, which is the first item to receive events the application generates. Normally the user decides the first responder. When the user touches a text field, the text field becomes the first responder, and the virtual keyboard appears for the user to enter text.

Object Library

Clicking the Objects button in the library selector bar opens the object library, which gives you access to each user interface element. Selecting an element from the object library opens a pop-up editor that provides a detailed description of the element. Click the Done button when you're finished viewing the description.

The object library initially shows every user interface element. Use the pop-up menu to show a subset of elements. I will explain the groups and the elements in each group shortly. Use the search field to find a particular element.

For more information on user interface elements, read Apple's *iOS Human Interface Guidelines*. The guidelines describe user interface elements as well as provide advice on when to use certain elements. The guidelines are part of Apple's documentation, which you can read in the Organizer.

Controls

The Controls group contains the controls you add to a view in an iOS application. This group contains the following controls:

- Label, which displays static text.
- Round rect button.
- Segmented control, which is a control that is divided into multiple segments. Each segment acts as a button.
- Text field, which you use to enter small amounts of text.
- Slider, which you use to set numeric values visually.
- Switch, which shows the Boolean state of a value.
- Activity indicator view, which provides feedback when a lengthy task is in progress.

- Progress view, which shows the progress of a lengthy task, such as downloading a file.
- Page control, which indicates what page you're reading in a multi-page document.
- Stepper, which increments or decrements a value.

Data Views

The Data Views group contains all the views that are subclasses of `UIView`. This group contains the following views:

- Table view, which displays a table.
- Table view cell, which you use to add custom cells to a table view. Table view cells should go in their own xib file.
- Collection view.
- Collection view cell.
- Collection reusable view.
- Image view, which displays an image.
- Text view, which allows the user to enter large amounts of text.
- Web view, which displays content from the World Wide Web.
- Map view, which displays map content.
- Scroll view, which displays content that is too large to fit in a window. Many of the items in the Data Views group inherit from `UIScrollView`.
- Date picker, which lets users select dates and times.
- Picker view, which displays a spinning wheel of values.
- Ad banner view, which displays ads in your application.
- GLKit view, which provides a view for drawing OpenGL ES content. The GLKit view requires iOS 5 or later and OpenGL ES 2.0.

Apple added the collection view, collection view cell, and collection reusable view in Xcode 4.5 and iOS 6.0. A collection view is similar to a table view, but you can customize the collection view's layout. Using a collection view allows you to have multi-column grids and tiled layouts. The collection view consists of collection view cells. A collection reusable view defines the behavior for the cells in the collection view. The collection reusable view allows you to use the same view for different types of content.

If you want a table view in your application, add a table view controller. The table view controller includes a table view. If you want a collection view in your application, add a collection view controller. The collection view controller includes a collection view. If you want a GLKit view in your application, add a GLKit view controller. The GLKit view controller includes a GLKit view. The section "Objects and Controllers" later in this chapter has more information on controllers.

Gesture Recognizers

Use the Gesture Recognizers group to recognize the following gestures: tap, pinch, rotation, swipe, pan, and long press. To add a gesture recognizer to your user interface, drag it to the canvas in a xib file. If you're using a storyboard, drag the gesture recognizer to a scene in the object list. Use the attributes inspector to configure a gesture recognizer.

Objects and Controllers

The Objects and Controllers group contains elements that are not visible in your user interface. This group contains the following items:

- Object, which you use to add instances of your application's classes to the xib file so you can connect user interface elements to your classes.
- External object, which is a proxy for an object that exists outside the xib file's contents. Use an external object to access an object in another xib file.
- View controller, which manages a view.
- Table view controller, which manages a table view.
- Collection view controller, which manages a collection view. Apple added the collection view controller in Xcode 4.5.
- Navigation controller, which manages hierarchical content.
- Tab bar controller, which manages a tab bar and its items.
- Page view controller, which manages multiple view controllers as pages.
- GLKit view controller, which manages a GLKit view for drawing OpenGL ES content.
- Split view controller, which manages two view controllers in a split view.

A collection view controller includes a collection view. A navigation controller includes a navigation bar, a navigation item, and a view controller. A tab bar controller includes a tab bar, two tab bar items, and a view controller for each tab bar item. A table view controller includes a table view. A GLKit view controller includes a GLKit view. If you want a collection view, navigation bar, tab bar, table view, or GLKit view in your application, add the appropriate controller. Adding the controller gives you the collection view, navigation bar, tab bar, table view, or GLKit view.

Apple added the split view controller to Interface Builder in Xcode 4.4. iPad applications are more likely to use a split view controller due to the iPad's larger screen size.

Windows and Bars

The Windows and Bars group contains, you guessed it, windows and bars. The following is a complete list of items in the Windows and Bars group:

- View
- Window
- Navigation bar
- Navigation item
- Search bar
- Search bar and search display controller
- Toolbar
- Bar button item
- Fixed space bar button item
- Flexible space bar button item
- Tab bar
- Tab bar item

The search bar and search display controller adds a controller for the search bar to the xib file. Using a search bar and search display controller saves you from having to add a controller manually.

If you want a navigation bar or tab bar in your application, add a navigation controller or tab bar controller. The navigation controller includes a navigation bar, and the tab bar controller includes a tab bar.

Use the navigation item to add an item to a navigation bar. Use the bar button item to add a button to a toolbar or a navigation bar. Use the flexible space and fixed space bar button items to add buttons to toolbars. Use the tab bar item to add a button to a tab bar.

Media Library

The media library contains audio, image, and video files. Choose View > Utilities > Show Media Library to display the media library. The media library has two sections: System and Workspace. The System section is empty for iOS projects. The Workspace section contains media files you have added to the projects in your workspace. If you haven't added media files to your projects, the Workspace section will be empty.

Selecting a file from the media library opens a pop-up editor for the file. You can play audio and video files from the pop-up editor and view image files.

Inspectors

Interface Builder has six inspectors for iOS applications. The inspector area has a selector bar at the top with a button for each inspector. The inspectors, running from left to right in the selector bar, are the following:

- File
- Quick Help
- Identity
- Attributes
- Size
- Connections

Some elements do not have all the inspectors. Controllers have an empty size inspector because they are not visible.

The file inspector applies to the xib file as opposed to a specific user interface element. The Quick Help inspector is rarely used in xib files.

File Inspector

The file inspector has one section specifically for xib files: the Interface Builder Document section. There are three things you can set in the inspector: deployment target, development target, and localization locking.

Use the Deployment menu to choose the deployment target, which is the earliest version of iOS that can load the xib file or storyboard file. If your interface has any problems on older versions of iOS, they appear in the issue navigator. Choose View > Navigators > Show Issue Navigator to open the issue navigator.

Use the Development menu to choose the development target, which is the earliest version of Interface Builder that can open the document. Setting the development target can help when working with other developers who are using older versions of Interface Builder.

Use the Localization Locking default pop-up menu to choose what parts of the xib file should be locked. Initially nothing is locked, which is good when you're starting to create the user interface. When you have the interface finalized, you may want to lock the document so you don't accidentally change something. If you're going to translate your application to other spoken languages, you would lock the non-localizable properties. By locking the non-localizable properties, translators can translate your application's titles and labels without messing up the rest of your interface.

Quick Help Inspector

Each element in the object library has a corresponding class. For example, a window is an instance of the `UIWindow` class. Selecting an element from a window or the object list provides information on the element's class in the Quick Help inspector as well as links to documentation on the class.

Identity Inspector

The identity inspector has the following sections for iOS applications: Custom Class, Identity, User Defined Runtime Attributes, Document, and Accessibility. Apple added the User Defined Runtime Attributes and Document sections in Xcode 4.5. Not every element has an Accessibility section.

Custom Class

Each user interface element in Interface Builder has a corresponding class. The Class combo box lets you set the class for an element. You must set the class if you create a subclass of an existing class and want to use that subclass in Interface Builder.

Identity

In Xcode 4.5 and later the Identity section contains a Restoration ID text field. The restoration ID works with application state preservation, which was added in iOS 6. Application state preservation allows you to restore your application's interface when the application goes into the background.

Give a user interface element a restoration ID to have its application state preserved. A restoration ID can be any string value. If you want to preserve the state of your application, it is very important to supply a restoration ID for each view controller. If a view controller has no restoration ID, none of its child views will be preserved, even if the child views have restoration IDs.

Earlier versions of Xcode place the contents of the Document section in the Identity section.

User Defined Runtime Attributes

The User Defined Runtime Attributes section lets you create a dictionary of key-value pairs for a custom object that does not have its own Interface Builder inspector. The dictionary is added to the object when your application launches.

Click the + button to add a key-value pair to the dictionary. Each key-value pair has three elements.

- Key Path, which is the key name.
- Type, which is the data type.
- Value, which is the key's initial value.

A key can have one of the following data types: Boolean, Number, String, Localized String, Point, Size, Rect, Range, Color, and Nil. The data type determines what you can enter for the value. The Nil data type doesn't let you enter anything. The Boolean data type gives you a checkbox. The Color data type provides a color picker to choose the color. The other data types use a text field. Double-click the value to enter the initial value for the key.

Document

The Document section makes things easier for you in Interface Builder. You can give an object a more descriptive name, a color label, and add notes to make the object easier to identify in the editor and jump bar. When you name an element using the Label text field, the element's name also changes in the object list.

You can also lock an element so other people can't make changes to it. If you're going to translate your application to other spoken languages, you would lock the non-localizable properties so translators can translate the object's text without messing up the rest of your interface.

Earlier versions of Xcode place the contents of the Document section in the Identity section.

Accessibility

The Accessibility section helps make your interface usable for people with disabilities, especially people with vision problems. Accessibility allows VoiceOver to provide descriptions of elements. Select the Accessibility Enabled checkbox to turn on accessibility for the user interface element.

Below the Accessibility Enabled checkbox are Label and Hint text fields. The label provides a concise description of an element. An Add button could have the label **Add**. The hint describes the results of an action. An Add button in a note taking application could have the hint **Add a note**.

Below the text fields are a series of Traits checkboxes that tell Xcode how to treat the element. Selecting the Keyboard Key checkbox tells Xcode to treat the element as a keyboard key.

Attributes Inspector

The attributes inspector is where you set an item's basic attributes, anything not covered by the other inspectors. Use the attributes inspector to perform the following tasks:

- Provide titles for controls.
- Set a slider's minimum, maximum, and initial values.
- Set the number of segments in a segmented control.
- Provide placeholder text for text views and text fields.
- Specify the keyboard to use for text fields and text views. If you have a text field to enter an email address, you can tell the text field to use the email address keyboard, which includes the @ character.
- Set a button's text color.
- Determine whether a window or view supports multi-touch events.

Each user interface element has its own set of attributes, which means what you can set in the attributes inspector depends on the item.

Size Inspector

The size inspector controls the size, position, and alignment of user interface elements.

Setting an Element's Size and Position

The View section of the size inspector has text fields to set an element's size and position. To reposition a view enter new values in the X and Y text fields. To resize a view enter new values in the Width and Height text fields.

Next to the text fields is the Origin field, which consists of a white square with nine dots. Clicking a dot tells you its position. If you click the dot in the lower left corner, the X and Y text fields display the coordinates of the lower left corner of the element.

Autosizing

The Autosizing section determines a view's position and size when the window's size changes. A window's size changes when the user changes the device's orientation from portrait to landscape and vice versa. The Autosizing section has one square inside a larger square, which you can see in Figure 4.4. A preview window appears next to the square. The preview shows you what happens to the view when the window changes size.

Springs

The inner square has two springs: one for vertical resizing and one for horizontal resizing. Clicking a spring toggles it. If a spring is not highlighted, the object maintains its size and position in the window. Controls like buttons and switches are items you wouldn't normally resize. You usually don't want your buttons to get huge when the window gets larger.

If a spring is highlighted (darker red), the object changes size when the window resizes. Views are the most common items to resize. If you're writing a text editor and the user rotates the device, the text view's size should reflect the rotation.

Struts

The outer square has four struts: one for each side of the view's rectangle. Clicking a strut toggles it. If a strut is not highlighted, the object maintains its relative position in the window. If a view is in the center of the window, it remains in the center of the window when the window resizes.

If a strut is highlighted (darker red), the object keeps its literal position in the window. If the lower left corner of a view is at the position (150, 100) in the window, it stays at (150, 100) when the window resizes.

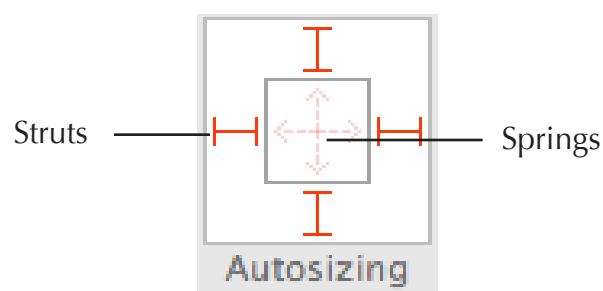


Figure 4.4

Autosizing square

Aligning Elements

Below the View section of the size inspector is the Arrange menu. This menu allows you to align multiple elements. Select the elements you want to align and make a choice from the Arrange menu. The menu has sections to align the elements vertically and horizontally. You have the following vertical alignment options:

- Align top edges
- Align vertical centers
- Align baselines
- Align bottom edges

You have the following horizontal alignment options:

- Align left edges
- Align horizontal centers
- Align right edges

Choosing Editor > Align accomplishes the same thing as using the Arrange menu.

Positioning Items in a Containing View

The Arrange menu also allows you to position items in their containing view. Select the element or elements you want to position and make a choice from the Arrange menu. You can center horizontally or vertically in the containing view. You can also fill the containing view horizontally or vertically, although filling a containing view is rarely necessary for iOS windows. Filling the containing view is normally done for larger views like table views and text views, and Interface Builder is set to easily make a large view fill the window.

Sizing Windows

Because iPhone and iPad windows take up the whole screen, you cannot change the size and position of windows in iOS. You can use the Origin field to show the x and y coordinates of the window's edges.

Connections Inspector

Cocoa Touch programs use messages to send data from one object to another. Connections let your user interface elements send messages to each other, send messages to model and controller objects, and receive messages from model and controller objects. The connections inspector is where you manage your user interface elements' connections. The connections inspector shows an element's outlets and actions.

Outlets are variables in a class that can connect to other classes or controls. The connections inspector has three sections for outlets: Outlets, Referencing Outlets, and Referencing Outlet Collections. The Outlets section shows you the selected element's outlets. The Referencing Outlets section lists any outlets in other elements that reference the selected element. The Referencing Outlet Collections section lists any outlet collections in other elements that reference the selected element. An outlet collection is a group of outlets that you apply to an array of elements.

The connections inspector has two sections for actions: Sent Events and Received Actions. The Sent Events section shows you the events the selected element can send to other objects. The Received Actions section shows you the actions the selected element can receive, actions other elements send to this element.

Remember that the connections inspector lists all the possible connections an element can have, not just the ones where you made a connection. If there is a connection, the outlet or action is highlighted with a destination, the connecting object, appearing next to it. The radio button next to the outlet or action is selected as well, which you can see in Figure 4.3. Click the X button next to the connecting object to remove the connection.

Connecting to Your Classes

One of Interface Builder's most powerful features is its ability to make connections from user interface elements to your application's classes. You can add outlets, outlet collections, and actions to your classes from Interface Builder for iOS applications. Adding connections to your classes requires the following steps:

1. Add an Objective-C class to your project.
2. Add an Object to the xib file.
3. Set the Object's class to your class.
4. Open the assistant editor.
5. Make connections to the header file.

If you haven't created your class, choose File > New > File. Select Cocoa Touch under iOS on the left side of the New File Assistant. Select Objective-C class. Click the Next button. Enter a class name and choose a subclass. Click the Next button. Pick a location to save the file and click the Save button.

Drag the Object item from the object library to the canvas to add an instance of your class to the xib file. Select the Object item from the jump bar and open the identity inspector. Choose your class from the Class combo box at the top of the inspector.

Opening the assistant editor allows you to have both the xib file and the header file visible so you can make connections to the header file. Click the center button in the Editor group in the project window toolbar to open the assistant editor.

To make a connection, right-click on an Interface Builder object (any UI element) and drag to the header file.

The interface for a newly created Objective-C class looks something like the following:

```
@interface MyClass : NSObject {  
  
}  
  
@end
```

If your drag destination is inside the braces of the `@interface` declaration, you can add an outlet or outlet collection to your class. If your drag destination is between the closing brace and the `@end` directive, you can add an outlet, outlet collection or action. Sometimes you can't add an action, depending on the item you drag to the header file. Creating an outlet or outlet collection outside the closing brace creates a property for the variable so you don't have to write accessor methods for the outlet or outlet collection.

When you create the connection, a pop-up editor opens. Use the Connection pop-up cell to choose between outlet, outlet collection, and action. For an outlet or outlet collection you can name it and specify its data type. For an action you can specify its name, return type, trigger event, and arguments. Click the Connect button to finish creating the connection.

If you make a connection to an existing outlet, outlet collection, or action in the header file, Interface Builder lets you connect the outlet, outlet collection, or action. But the ability to make a connection depends on the user interface element and the outlet, outlet collection, or header.

Grouping Objects

Although iOS applications have less need to group user interface elements than Mac applications, Interface Builder provides the ability to group iOS elements. Select the objects in the canvas that you want to group together. Choose Editor > Embed. You can embed objects in a view, a scroll view, a navigation controller, or a tab bar controller. If you want to unembed what you embedded, select the embedded view and choose Editor > Unembed.

Storyboarding

Storyboarding is a new way of creating user interfaces for iOS applications. Instead of using multiple xib files to build your user interface, storyboarding places all your application's screens together. Storyboarding allows you to create your entire interface in one place. Storyboarding also allows you to create prototype-based tables in Interface Builder. These tables let you build the table and its cells in Interface Builder without having to supply a data source.

Apple added storyboarding support in iOS 5. You must continue to use xib files if you want to support earlier versions of iOS.

Storyboards consist of scenes and segues. A *scene* represents one screen of content. A *segue* (segueway) is a transition from one scene to another. Examples of segues include showing a modal view, showing a popover, and pushes. A push uses a navigation controller to slide a view onto the scene.

Creating a Storyboard

When you create an iOS application project, you have the option to use storyboarding. Select the Use Storyboard checkbox to use a storyboard. If you have an existing project you want to storyboard, add a storyboard file to the project. Choose File > New > File. The storyboard file is in the User Interface section under iOS. Choosing File > New > File creates a blank storyboard.

Storyboard files have the extension **.storyboard**. Select the storyboard file from the project navigator to open the storyboard canvas.

The Storyboard Canvas

Figure 4.5 shows an example of a storyboard canvas. If you add a storyboard to an existing project, the canvas is blank. The lower left corner of the canvas has a button that toggles showing the object list. The lower right corner of the canvas has buttons to zoom the canvas. Zooming in allows you to focus on a scene. Zooming out provides an overview of the scenes and segues in the storyboard.

If you look at the two views shown in Figure 4.5, you will see the dock at the bottom of the left view. The dock shows the scene's objects. If you need an instance of one of your application's classes in a scene, drag an Object from the object library to the dock. Select the view controller in the object list to show the dock.

Working with Scenes and Segues

Add a scene to the storyboard by dragging a controller from the object library to the canvas. Creating a segue is similar to creating a connection between user interface elements. Select a view controller, right-click, and drag to a second view controller. A HUD window opens with a list of segue types. Select a segue type from the list. Use the attributes inspector to change the segue type and name the segue so you can access it in your code. If you add a modal segue, choose a transition from the Transition menu in the attributes inspector. Select the Animates checkbox if you want the modal segue to be animated. If you add a custom segue, enter the custom segue's class in the Segue Class text field. The class must be a subclass of `UIStoryboardSegue`.



Figure 4.5

Storyboard canvas

Creating a Table in Interface Builder

If you use a storyboard, you can use prototype cells to build a table in Interface Builder. Building the table in Interface Builder gives you a better idea of how your table will look before you build and run the application.

When creating a table the first step is to add the table view. Drag a table view controller from the object library to the canvas. Add table view cells to the table by dragging them from the object library to the table view. Use the Identifier text field in the attributes inspector to name the cells so you can access them in your code. Drag an element from the object library to the table view cell to add the element to the cell. A label is the most common element to add to a table view cell.

To create static content for a table, open the table view's attributes inspector. Choose Static Cells from the Content menu. Use the attributes inspector to set the number of sections in the table view and the table view's style.

Subclassing `UITableViewController` and `UITableViewCell` can make table creation easier in Interface Builder. If you subclass `UITableViewController` or `UITableViewCell`, use the identity inspector to set the custom class to your subclass.

Auto Layout

Auto layout maintains relationships between views. Auto layout can keep views from getting too large, too small, too close to other views, and too close to the edges of the window. Suppose you have two buttons next to each other in a window. You change the text of the left button so the text is twice as long. Before auto layout you would have to make the following changes manually: resize the left button to fit the new text and move the right button to maintain its position with the resized left button. With auto layout the left button resizes automatically to reflect the new text, and the right button moves to the proper position. Auto layout makes localizing user interfaces much easier.

Apple added auto layout in iOS 6.0. Auto layout does not work on earlier versions of iOS. If you use auto layout the xib or storyboard file will not load in earlier versions of iOS.

Turning on Auto Layout

If you create a new iOS application project, you don't have to worry about turning on auto layout. But if you have an existing project that you want to convert to auto layout, you must enable it. To enable auto layout, open the file inspector for the xib or storyboard file. Select the Use Autolayout checkbox in the Interface Builder Document section of the file inspector.

When you turn on auto layout, the autosizing section of the size inspector is removed. Auto layout replaces the autosizing mask.

Constraints

When you enable auto layout you won't notice any difference in the xib or storyboard file until you select something inside the window. Selecting a control shows the control's constraints, which look similar to the struts in the size inspector's autosizing section. Constraints are objects that express relationships between views.

Figure 4.6 shows an example of the constraints when you select a button. There are four constraints in the example: two for the Test button and two that involve both buttons. The constraints for the Test button keep the button from getting too close to the left edge and the bottom of the window. The constraints that involve both buttons keep them from getting too close to each other and keep their baselines aligned.

Editing Constraints

To modify a constraint open the attributes inspector and select the constraint. You may find it easier to use the jump bar to select a constraint. Click the View item in the jump bar and choose Constraints to open a menu of constraints. What you can control depends on the constraint, but most constraints have three fields: a Relation pop-up menu, a Constant text field, and a Priority slider.

The Relation menu has three possible values: Equal, Less Than or Equal, and Greater Than or Equal. The horizontal space between two controls is an example of a constraint that would have an equal relation. A less than or equal relation keeps views from getting too large or getting too far away from another view or the edge of the window. A greater than or equal relation is typically used with the height and width of large views like text views. They ensure the view doesn't get so small that it becomes unusable when the user rotates the device.

The Constant text field stores the value the Relation menu uses. Suppose you have a text view and you want it to always be at least 200 pixels wide. You would set the relation to greater than or equal and enter 200 in the Constant field.

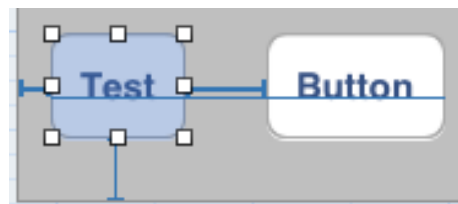


Figure 4.6

Constraints for a selected button

Some constraints, usually ones that enforce space constraints, have a Standard checkbox next to the Constant text field. Selecting the Standard checkbox tells Interface Builder to use Apple's standards, which keeps you from having to know how much space there should be between two buttons or having to know how close to the edge a control should be. When you select the Standard checkbox, the Constant text field says Auto.

The Priority slider measures when the constraint should be enforced. The value of the priority can range from 0 to 1000. 1000 means always enforce the constraint. 0 means every other constraint or action, such as the user rotating the device, takes precedence over the constraint. When you drag the slider, a popover opens that explains what the priority means in practice.

Content Priorities

If you select a user interface element and open the size inspector, you will see two sets of sliders: content hugging priority and content compression resistance priority. Each set has two sliders: horizontal and vertical. The value for each slider can range from 0 to 1000, with 1000 being the highest priority.

The content hugging priority measures how much the edges of the element hug the element's content. The higher the content hugging priority, the less likely the element will resize. A button normally has high vertical content hugging priority and lower horizontal content hugging priority. Horizontal padding on a button is more acceptable than vertical padding.

The content compression resistance priority measures how likely the element's content is to be clipped or compressed. The higher the priority, the less likely the element's content will be clipped. A button should have higher content compression resistance priority than a text view. Clipping a button's title can make it unusable while clipping a text view's content allows the user to read at least some of the text in the view.

Adding Constraints

When you place user interface elements in a window, Xcode automatically creates constraints based on where you place the items. But you can also create your own constraints by using the Editor menu. Choosing Editor > Pin is the most common way of adding a constraint to a single view. Choosing Editor > Align is the most common way of adding constraints to a pair of views.

Xcode 4.5 adds a set of three buttons to the lower right corner of the canvas to make working with auto layout easier. The left button aligns and centers views. The center button pins a view's size and spacing. The right button lets you determine how constraints are applied when resizing views. You can apply constraints to the resized view's descendants or to its siblings and ancestors.

Constraints you create and constraints you edit in the attributes inspector have thicker lines than the constraints Interface Builder automatically creates.

Chapter 5

Modeling Tools

Xcode has two modeling tools to help you create Mac and iOS applications. The data modeling tool lets you create data structures visually without having to write code. The mapping tool lets you create a mapping model to migrate data from an older data model to a new one.

Xcode's modeling tools require Core Data. If you're not using Core Data, you can skip this chapter and come back to it later.

Data Models

Xcode's data modeling tool lets you create your program's data visually instead of writing code. The data modeling tool uses the Core Data framework. Core Data is a huge topic, too large for me to cover in this book. Read the *Core Data Programming Guide* that is part of Apple's documentation to learn more about Core Data.

There are three terms you must know before you can model data: entities, attributes, and relationships. Entities are the basic building blocks of your data models. Entities consist of attributes and relationships. Attributes contain data. Relationships represent relationships to other objects. In programming terms, entities are your classes, attributes are your data members, and relationships are your methods.

Adding a Data Model File to Your Project

When you create a Core Data application project, Xcode adds a data model file for you. If you have an existing project and want to use the data modeling tool, you must add a data model file to your project.

1. Choose File > New > File.
2. Select Data Model from the file list and click the Next button. The data model file is in the Core Data group under both Mac OS X and iOS. Choose the appropriate one.
3. Name the file, select the targets you want to add the data model to, and click the Create button. Make sure the data model is added to the application target.

XML Data Models

Xcode 4 uses XML data model files. Previous versions of Xcode used binary data model files. Because XML data model files are text files, they have two advantages over binary data model files. First, you can view the differences between two versions of a data model file in Xcode's version editor or other diff tool. Second, XML files work better with version control systems than binary files.

If you create a Core Data Xcode project, the data model file is an XML file. Those of you with existing projects can convert your data models to XML files from the file inspector. Select the data model file from the project navigator. If your data model has multiple versions, make sure you select the data model file, the file with extension `.xcdatamodel`, and not the bundle that has the extension `.xcdatamodeld`. In the Core Data Model section of the file inspector is the Minimum Tools Version menu. Choose Automatic from the menu, which sets the minimum version to the version of Xcode you're running. Setting the minimum version to Xcode 4.1 or later converts the data model to an XML data model.

Data Model Editor

Select the data model from the project navigator to open the data model editor. The data model editor, shown in Figure 5.1, is where you create your data models. The window has the following sections:

- Top-level components
- Detail area
- Bottom area
- Graph view

When working with a data model, you should have the utility area open so you can access the data model inspector. The data model inspector is where you edit most settings in your data model. Choose View > Utilities > Show Data Model Inspector to open the data model inspector.

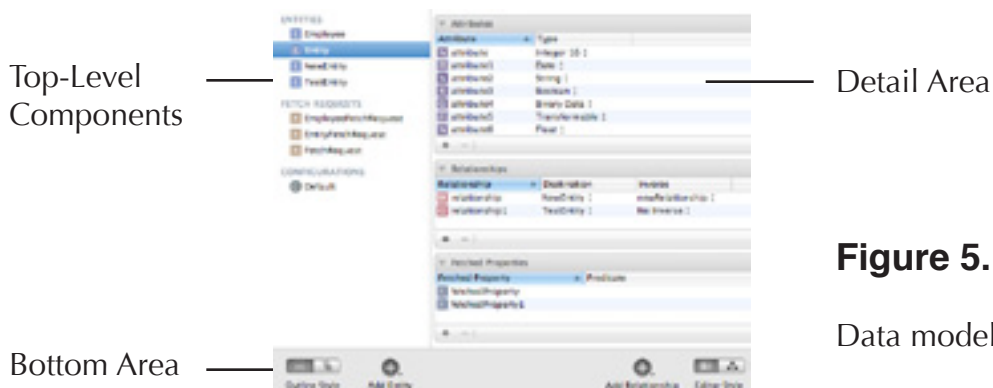


Figure 5.1

Data model editor

Top-Level Components

The top-level components area contains a list of your data model's entities, fetch requests, and configurations. Read the "Adding Fetch Requests" section later in this chapter for more information on fetch requests. Read the "Adding Configurations" section later in this chapter for more information on configurations.

Detail Area

Selecting an entity from the entity list fills the detail area with a list of the entity's attributes, relationships, and fetched properties. The attribute list shows each attribute's name and data type. The relationship list shows each relationship's name, destination, and inverse relationship. The fetched property list shows each fetched property's name and predicate. Each of the lists has buttons to add and remove items.

Selecting a fetch request from the fetch request list fills the detail area with a predicate editor, which allows you to specify the conditions for the fetch request.

Selecting a configuration from the configuration list fills the detail area with a list of the entities that make up the configuration. For each entity the list tells you the entity's name, the class it belongs to, and whether or not the entity is abstract. The class an entity belongs to must be `NSManagedObject` or a subclass of `NSManagedObject`.

Abstract entities are entities you don't create instances of in your application. They are meant to be inherited by other entities. An abstract entity is similar to an abstract class in object-oriented programming, such as Cocoa's `NSObject` class. You don't create objects in your Cocoa applications; you create the objects that inherit from `NSObject`: windows, views, controls, strings, arrays, etc.

Bottom Area

The bottom of the data model editor has a set of four buttons.

- Outline style button, which displays a flat or hierarchical list of entities. Your data model must have entities that inherit from other entities to see any difference between a flat list and a hierarchical list.
- A button to add an entity, fetch request, or configuration.
- A button to add an attribute, relationship, or fetched property.
- Editor Style button: table or graph view. Click the right button to switch to the graph view.

Graph View

The graph view, which you can see in Figure 5.2, shows the entities and their relationships to each other. Lines represent the relationships between entities. The arrow points to the destination. Lines with arrowheads on both ends indicate a bidirectional relationship, which means the relationship has an inverse relationship. A line with a single arrowhead indicates a to-one relationship, which means there is only one destination object. A line with a double arrowhead indicates a to-many relationship, which means there can be multiple destination objects.

Each entity in the diagram has at least three compartments: the entity name, attributes compartment, and relationships compartment. If an entity has fetched properties, there will be a fourth compartment for the fetched properties. Use the disclosure triangles to show and hide the attributes, relationships, and fetched properties in their respective compartments.

Adding Entities

To add an entity to your data model, click the Add Entity button at the bottom of the data model editor. The Add Entity button may have the caption Add Fetch Request or Add Configuration. The same button lets you create entities, fetch requests, and configurations. After adding an entity, use the Entity section of the data model inspector to specify the following information about the entity:

- Its name.
- Its class. The class must be `NSManagedObject` or a subclass of `NSManagedObject`.
- The entity's parent.
- Whether or not the entity is abstract.
- Indexes.

When you're starting out, giving each entity the class `NSManagedObject` keeps things simple. Later on you may want to give each entity its own class. Each of these classes will be subclasses of `NSManagedObject`.

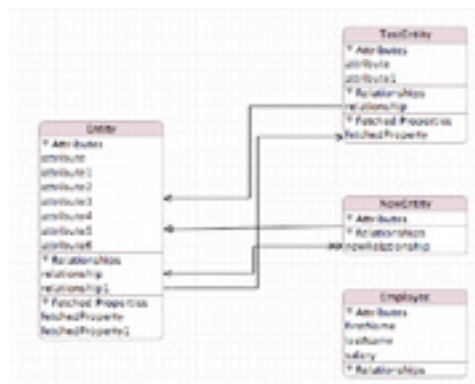


Figure 5.2

Data model
graph view

Abstract entities are never instantiated. They are meant to be parents of concrete entities.

Use the Indexes list to add compound indexes to the entity. A compound index is an index across multiple properties, like attributes and relationships. Using a compound index can make searches faster. You must be using the SQLite store to add compound indexes. Click the + button at the bottom of the Indexes list to add a set of indexes. Separate the indexes with commas. The names of your attributes and relationships are the most common indexes.

To delete an entity, select it from the entity list and press the Delete key.

Adding Attributes

To add attributes to an entity, select the entity from the entity list. Click the + button under the attributes list.

Setting an Attribute's Name and Data Type

Use the attributes list to set the attribute's name and data type. An attribute can have the following data types:

- Undefined
- Integer 16
- Integer 32
- Integer 64
- Decimal
- Double
- Float
- String
- Boolean
- Date
- Binary Data
- Transformable

Most of the data types correspond to C and Objective-C data types. The data types that require more explanation are Undefined, Binary Data, and Transformable. Transient attributes should have an undefined data type. Only transient attributes can have an undefined data type.

Give an attribute a data type of binary data when the attribute is more complicated than any of the Objective-C data types, such as a data structure. You should also store large data sets as binary data, such as audio and video data.

Attributes with a transformable data type are converted to and from instances of `NSData`. Suppose you have an attribute that stores some text you want to display in a text view. You're going to display rich text (styled text) in the text view and use `NSTextView`'s Attributed String binding. If you make the attribute a string, you won't be able to use the Attributed String binding. By making the attribute transformable, you can use the Attributed String binding and display rich text.

Setting Additional Attribute Information

Use the data model inspector to set additional properties of the attribute. You can specify whether the attribute is optional, whether it is transient, and whether it is indexed. If an attribute is not optional, it must have a value or your application will get errors when the user tries to save.

Core Data automatically stores your data in a data file and retrieves the data from the data file. Transient attributes are not stored in the data file.

Indexed attributes have an index created for them. Indexes make finding a record in the Core Data store faster, but they also increase the size of the store. If you have a Core Data store of customers, the customer's name would be a good attribute to index, but the customer's street address would be an unnecessary attribute to index.

When you specify the data type, what you can set depends on the data type you chose. For numerical and date attributes you can set the attribute's minimum, maximum, and default values. For Boolean attributes you can set the default value. For a string you can set the minimum length, maximum length, default value, and a regular expression. Use a regular expression to constrain the values the string can have. If you were using a string attribute to store a number, such as a credit card number, you would use a regular expression to limit the attribute to storing the characters 0–9.

If the attribute's data type is binary data, you will see an Allows External Storage checkbox. Selecting the Allows External Storage checkbox tells Xcode to store the attribute outside the Core Data store. If the attribute stores a large amount of data, such as a video, you should select the Allows External Storage checkbox.

Adding Relationships

To add relationships to an entity, select the entity from the entity list. Click the + button under the relationships list. You can set the following properties for a relationship:

- Name.
- Destination, which is one of the entities in the data model. Every relationship requires a destination. If you do not supply a destination entity, you will get an error when you try to build the project.
- Inverse, which bidirectional relationships use. If you have a relationship from entity A to entity B, the inverse is the corresponding relationship from B to A. Most relationships should have an inverse relationship.
- Optional. An optional relationship does not require an object to exist in the destination.
- Transient. Transient relationships are not stored in the data file Core Data uses to store your data.
- To-Many. When you have a to-many relationship, the destination can have more than one object. An example of a to-many relationship is a student enrolling in courses. A student can enroll in more than one course.
- Ordered, which you can use to assign positions in to-many relationships.
- Min and Max Counts let you set the minimum and maximum number of destination objects in the relationship.
- Delete rule.

You can set a relationship's name, destination, and inverse from the relationships list. Use the data model inspector to set the other relationship properties.

If you don't select the To-Many Relationship checkbox, the relationship is a one-to-one relationship. You cannot set the minimum and maximum counts for one-to-one relationships.

An ordered relationship assigns positions in a to-many relationship, which means you must have a to-many relationship to create an ordered relationship. Create an ordered relationship when the objects in a relationship must be in a specific order that is not tied to the value of an attribute in the data model. Displaying a list of chapters in a book is an example of an ordered relationship: Chapter 1 should appear before Chapter 2, Chapter 2 should appear before Chapter 3, and so on. If you sorted the chapters alphabetically by their title, they would be out of order. An ordered relationship shows the chapters in the proper order.

Delete rules determine what happens to the source and destination objects of a relationship when you delete the source object. There are four delete rules.

- No Action. Delete the source object but don't delete the destination objects.
- Nullify. Delete the source object. Do not delete destination objects, but set each destination object's inverse relationship to NULL.
- Cascade. Delete the source object and all destination objects.
- Deny. Do not allow the deletion if the source object has destination objects.

Adding Fetched Properties

A fetched property is a special type of relationship. Supply a set of conditions. The fetched property contains the related objects that meet the conditions you supply.

To add fetched properties to an entity, select the entity from the entity list. Click the + button at the bottom of the fetched properties list.

Enter a predicate in the Predicate column of the fetched properties list. A predicate is a set of conditions. Writing a predicate is similar to writing an `if` statement in C, C++, or Objective-C. For more information on predicates, read Apple's *Predicate Programming Guide*. The *Predicate Programming Guide* is part of the Mac OS X and iOS documentation, which you can read in Xcode.

Use the data model inspector to set a destination entity for the fetched property. Choose a destination entity from the Destination pop-up menu.

Adding Fetch Requests

A fetch request is an object that tells Core Data the data you want to find. Create a fetch request to retrieve data from an entity.

To add a fetch request, click the Add Entity button in the lower left corner of the data model editor and choose Add Fetch Request. The button may say Add Configuration. The same button lets you create entities, fetch requests, and configurations.

Editing the Fetch Request's Predicate with the Predicate Builder

Predicates are the way you specify conditions when searching. Select a fetch request from the left side of the data model editor to open the predicate builder, which you can see in Figure 5.3. If you prefer to create the predicate textually, click the right button on the right side of the predicate builder.

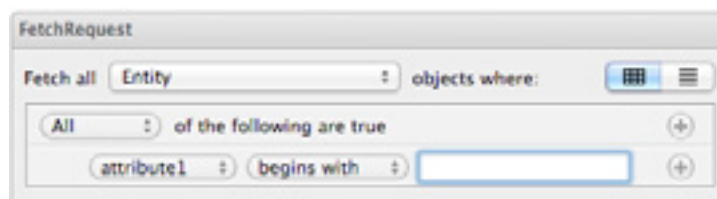


Figure 5.3

Predicate builder

The predicate builder initially has no conditions, but has a pop-up cell with the label of the following are true. The menu has the following values: All (And), Any (Or), and None (Not). This menu is especially important if you create compound predicates, which consist of multiple conditions.

To add a condition, click the + button. A pop-up cell opens with the name Expression. You can enter a condition in the text field, but in most cases the predicate involves one of the entity's attributes. Choose an attribute from the Expression menu. Choosing an attribute opens a second pop-up cell.

The second pop-up cell contains conditions. The conditions depend on the attribute's data type. Numerical attributes have mathematical conditions like greater than, less than, and equal. String attributes have additional conditions such as begins with, ends with, and contains. Use the text field to specify the value.

Suppose you have an Employee entity with three attributes: `firstName`, `lastName`, and `salary`. You want to find the employees whose last names start with J and earn more than \$50,000 a year. To create this predicate, perform the following steps:

1. Click the + button to add a condition.
2. Choose `lastName` from the Expression pop-up menu. This creates another pop-up menu.
3. Choose begins with from the second pop-up menu.
4. Enter J in the text field.
5. Click either of the two + buttons to add a second condition.
6. Choose `salary` from the Expression pop-up menu. This creates another pop-up menu.
7. Choose is greater than from the second pop-up menu.
8. Enter 50000 in the text field.

Data Model Inspector for Fetch Requests

Use the data model inspector to set everything for the fetch request besides the predicate. The data model inspector has the following fields:

- Name.
- Entity, which is one of the data model's entities.
- Result Type, which is what the fetch request returns. The fetch request can return managed objects, managed object IDs, or dictionaries.
- Fetch Limit, which is the maximum number of objects the fetch request returns. Zero means no limit.
- Batch Size. If you set a non-zero batch size, Core Data retrieves the size number of objects from the store in one fetch. Suppose you set the batch size to 5 and have 50 objects in the store. Core Data retrieves the objects 5 at a time, using 10 fetches to fetch all the objects.

Advanced Checkboxes

The data model inspector for a fetch request has a series of Advanced checkboxes.

- Include Subentities. If you select this checkbox, the fetch request includes all of an entity's subentities.
- Include Property Values. If you deselect this checkbox, the fetch request obtains only enough information to identify the object. Not including property values in a fetch request reduces memory overhead, but only deselect the checkbox if you're sure you won't need the property data.
- Return Objects as Faults. If you select this checkbox, the objects returned are not pre-populated with their property values.
- Include Pending Changes. If you deselect this checkbox, the fetch request returns only objects that are in the persistent store. It does not check for unsaved changes.
- Return Distinct Results. If you select this checkbox, the fetch request returns only distinct values for specified fields. Use the method `setPropertiesToFetch:` to specify the fields. You must set the Result Type to Dictionaries in the data model inspector to be able to return distinct results.

Setting Information Dictionary Entries

All elements except fetch requests can have an information dictionary. This dictionary consists of key-value pairs. The information dictionary lets views and controllers access the model's properties. Values used by a fetched property's predicate and version information are examples of data that are placed in information dictionaries.

To modify the information dictionary, open the data model inspector. The User Info section in the data model inspector contains the information dictionary for the selected entity, attribute, relationship, or fetched property. Click the + button to add a dictionary entry. Give the entry a key and a value.

Adding Configurations

A configuration is a collection of entities. Configurations let you store entities in different Core Data stores. If Core Data didn't have configurations, you would be limited to one Core Data store that contains the entire data model. Xcode creates a default configuration for you that contains all the entities in your data model. Create a configuration only if you need multiple Core Data stores.

To create a configuration, click the Add Entity button in the lower left corner of the data model editor and choose Add Configuration. After creating the configuration, the next step is to add entities to it. Select an entity from the entity list and drag it to the configuration or drag it to the configuration's list of entities.

To delete a configuration, select it from the configurations list and press the Delete key.

Versioning

When you select an entity, attribute, or relationship in the data model editor, the data model inspector displays a Versioning section. The Versioning section has two text fields that help you handle changes to your data model's entities. The Hash Modifier text field lets you enter a string that marks the entity as being a new version. Enter a version hash modifier if you make a change to an entity that doesn't change the structure of the data model.

Suppose you have an attribute in your data model measuring customer satisfaction on a scale of 0 to 10. You decide to change the customer satisfaction scale so it ranges from 0 to 100. The structure of the data model is identical in both versions; they both store an integer. But what the data model represents has changed. By entering a version hash modifier, you let Core Data know you have a new version of the data model. You can enter any string you want as a version hash modifier.

The Renaming ID text field lets you specify a renaming identifier that resolves naming conflicts. Suppose you have a Customer entity and you change the name to Client. Giving the Client entity a renaming identifier of Customer helps Core Data resolve any conflicts between Customer and Client entities in the data model.

Advanced Checkboxes for Attributes and Relationships

When you select an attribute or relationship in the data model editor, the data model inspector shows two Advanced checkboxes: Index in Spotlight and Store in External Record File. If you select the Index in Spotlight checkbox, Core Data makes the property available to Spotlight. When someone using your application enters something in the Finder's search field that matches the property, any files that store the property show up in the search results.

Selecting the Store in External Record File checkbox tells Core Data to store the property in a separate file.

Synchronizing Data Models

Xcode 4.4 removed support for synchronizing data models from the data model inspector. You can ignore this section if you're using a newer version of Xcode.

The Synchronizing section is used to sync your data models with other applications and other devices, like laptop computers, cell phones, and iPods. To take advantage of synchronization, you must add the Sync Services framework to your project. Only entities, attributes, and relationships can be synchronized.

For more information on synchronizing data, read the *Sync Services Programming Guide*, which is part of Apple's documentation. The *Sync Services Programming Guide* has a section on syncing Core Data applications.

Syncing an Entity

To sync an entity, select it from the entity list. In the Entity Sync section of the data model inspector is a Synchronization pop-up menu. Choose Enabled from the menu.

When syncing an entity, the only mandatory step is to specify a data class using the Data Class combo box. The data class takes the following form:

```
com.CompanyName.AppName.EntityName
```

If the entity has a parent relationship, choose the parent from the Parent pop-up menu. A parent relationship is the name of a relationship that contains records belonging to the entity. The parent relationship displays a readable entity name if syncing causes conflicts.

The Exclude From Change Alert checkbox determines what happens when the entity's data changes. If the checkbox is not selected, an alert opens when the entity's data changes. You should limit data change alerts to the most important entities in your data model.

Syncing an Attribute

To sync an attribute, select it from the attribute list. In the Attribute Sync section of the data model inspector is a Synchronization pop-up menu. Choose Enabled from the menu.

Selecting the Identity Property checkbox tells Core Data to use that attribute to identify the record. The Exclude From Change Alert checkbox works for attributes like it does for entities. Deselect the checkbox if you want an alert to appear when the attribute's value changes.

Below the checkboxes are two pop-up menus that are used to automatically resolve conflicts during syncing. The Client Type menu determines what client Sync Services uses first for the syncing. There are four choices.

- Prefer App, which is your application.
- Prefer Device, which is the physical device, such as a laptop, cell phone, or iPod.
- Prefer Server, which is a server like Apple's iCloud.
- Prefer Peer, which is a peer-to-peer client.

The Record menu determines what record should be used for syncing. There are three choices.

- Prefer Truth, which means use the record in the truth database. The truth database contains all the client's records.
- Prefer Client, which means use the record on the client you chose from the Client Type menu.
- Prefer Last Modified, which means use the most recently modified record.

Syncing a Relationship

To sync a relationship, select it from the relationship list. In the Relationship Sync section of the data model inspector is a Synchronization pop-up menu. Choose Enabled from the menu.

Selecting the Identity Property checkbox tells Core Data to use the relationship to identify the record. The Exclude From Change Alert checkbox works for relationships like it does for entities and attributes. Deselect the checkbox if you want an alert to appear when the relationship's value changes.

Creating Source Code

Xcode can create class files for your data model's entities and create accessors for attributes and relationships. The accessors take the form of Objective-C 2.0 properties.

To create a class file for an entity, select it from the entity list and choose Editor > Create NSObject Subclass. A Save panel opens, which should be set to your project folder. Click the Create button to create the file. In the Save panel you will see a Use scalar properties for primitive data types checkbox. Selecting this checkbox tells Xcode to create scalar accessors instead of `NSNumber` objects. Suppose you have an attribute that is a 32-bit integer. A scalar accessor defines the attribute as a 32-bit integer instead of a `NSNumber` (`NSNumber` inherits from `NSNumber`) object. Scalar accessors allow you to skip the overhead of creating the `NSNumber` object.

To create an accessor, select an attribute or relationship. Choose Edit > Copy. Open the file where you want the code to appear. Select the location where you want the code to appear. Paste the code by choosing Edit > Paste ItemType, where ItemType is one of the following:

- Attribute Name
- Attribute Interface
- Attribute Implementation
- Relationship Name
- Relationship Interface
- Relationship Implementation

When creating accessors you must paste twice to fully create the accessors: once in the header file and once in the implementation file. Place the cursor outside a method in the implementation file. Place the cursor between the class's closing brace and the `@end` statement in the header file. If you place the cursor in any other location, all you will be able to paste is the name of the attribute or relationship.

Mapping Models

Mapping models are used to migrate data from one version of a data model to another. They specify the transformations needed to migrate the data. Why would you use a mapping model? Suppose you've written version 1.0 of a Core Data application. Time has passed and you write version 2.0. In the process of writing version 2.0, you make changes to the data model. By using a mapping model, people who saved data using version 1.0 of your application will be able to load their data in version 2.0.

If you're new to Xcode's modeling tools you won't be using mapping models right away because you have to create the initial data model first. When you need to make changes to the initial model, create a mapping model to migrate your data from the initial data model to the new model.

NOTE

Mac OS X 10.6 added the ability to perform simple data migrations automatically without you having to create a mapping model. iOS also has the ability to migrate data automatically. Because I'm covering Xcode's modeling tools in this chapter, I'm not covering automatic data migration.

Versioned and Non-Versioned Data Models

To create a mapping model, you must have a versioned data model. A versioned data model lets you save multiple versions of the data model. The data model's file extension lets you know whether it's versioned or non-versioned. A non-versioned data model is a single file with the extension `.xcdatamodel`. A versioned data model is a bundle with the extension `.xcdatamodeld`.

When you create a Core Data application project in Xcode 4, Xcode creates a versioned data model for you. If your data model happens to be non-versioned, choose Editor > Add Model Version to convert a data model to a versioned data model.

Adding a New Version of Your Data Model

Before you can use a mapping model, you must create a new version of your existing data model. A mapping model requires at least two versions of your data model to perform the mapping. Select the data model in the project navigator and choose Editor > Add Model Version. Name the new version of your data model, choose an old version from the Based on model pop-up menu, and click the Finish button.

Inside the bundle you will see at least two versions of the data model: the original and a copy of the data model titled `Filename.xcdatamodel`, where `Filename` is the name you gave the data model when you created a new version. Open the new version of the data model. Make the changes you want to make to the current data model. The changes involve adding, deleting, or editing the entities, attributes, and relationships in the data model.

To set the current data model version, make sure the file inspector is open. Select the `.xcdatamodeld` file in the project navigator. Use the Current pop-up menu in the Versioned Core Data Model section of the file inspector to set the current version. Make sure the current data model is the one where you made your changes to the data model.

Adding a Mapping Model to Your Project

After making the changes to the data model, you can add a mapping model to your project. Choose `File > New > File`. The mapping model file is in the Core Data section under both Mac OS X and iOS. Choose the appropriate one. Click the Next button.

Now it's time to select the source and target models for the mapping model. The source model is the model version you didn't modify, the original version. The target model is the model you modified, the new version. Remember that the source model is the old version of the data model, and the target model is the new version. After selecting the source and target models, name the mapping model and click the Create button.

After creating the mapping model, open it to make sure Xcode mapped the data from the old data model to the new data model properly. The mapping should work for relatively simple changes like the following:

- Adding entities, attributes, and relationships.
- Removing entities, attributes, and relationships.
- Renaming entities, attributes, and relationships.
- Making minor changes to an attribute's data type, such as changing an attribute from a 16-bit integer to a 32-bit integer.

Mapping Model Editor

Select the mapping model from the project navigator to open the mapping model editor, which you can see in Figure 5.4. On the left side is the entity mappings list. On the right side are attribute and relationship mappings. The bottom of the editor contains a button to add an entity mapping as well as pop-up menus for the source and destination data models.

When working with a mapping model, you should have the utility area open so you can access the mapping model inspector. The mapping model inspector is where you make most changes to mappings. Choose View > Utilities > Show Mapping Model Inspector to open the mapping model inspector.

Entity Mappings

The left side of the mapping model editor contains entity mappings. There is one listing for each entity in the source and destination data models. When you select an entity mapping from the entity mappings list, the mapping model inspector shows the following information:

- Mapping name. If the entity appears in both models, the mapping name is EntityToEntity, where Entity is the name of the entity. Otherwise, the mapping name is the name of the entity.
- Source, which is the entity name in the source data model. If you add an entity in a new version of a data model, there will be no source.
- Destination, which is the entity name in the destination data model.
- Type, which describes the type of mapping. Examples of mapping types include add, copy, and transform.
- Custom Policy, which is the name of the class for the mapping. It's a subclass of `NSEntityMigrationPolicy`.
- Source Fetch, which determines how Core Data fetches the data from the source data model.

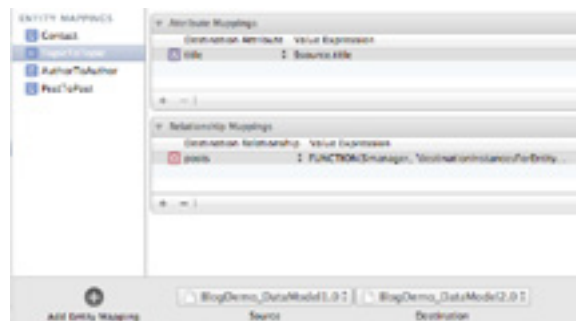


Figure 5.4

Mapping model editor

The mapping model inspector has two source fetch options: default and custom. If you choose the default source fetch, you can supply a filter predicate. A filter predicate is like the predicates you can create for fetched properties and fetch requests in a data model. If you choose a custom source fetch, you must supply a source expression for the custom fetch in the Source Expression text field. A source expression is any expression that evaluates to a collection of managed objects.

Property Mappings

Next to the entity mappings section are the property mappings. Selecting an entity fills the mapping model editor with the mappings for the entity's attributes and relationships. There are two columns of information for property mappings: Destination Attribute or Relationship, and Value Expression, which represents an expression in a predicate.

New properties have a blank value expression. Attributes in the source data model have the following value expression:

```
$source.AttributeName
```

Relationships in the source model have a value expression that looks similar to the following:

```
FUNCTION($manager, "destinationInstancesForEntityMappingName:sourceInstances:", "RelationshipNameToRelationshipName", $source.RelationshipName)
```

Use the + and minus buttons to add and remove attribute and relationship mappings.

Changing Attribute Mapping Data

If you select an attribute from the attribute mappings list, you can change the value expression. Changing a property's value expression is optional. Value expressions can be as complicated as you want to make them, but you should stick to simpler value expressions in the mapping model. If you need a complicated value expression to migrate an attribute, write code to handle the migration.

A value expression has the data type `NSExpression`. Read the *NSExpression Class Reference* for more detailed information on value expressions. The *NSExpression Class Reference* has a link to Apple's *Predicate Programming Guide*, which is also helpful.

Changing Relationship Mapping Data

If you select a relationship, you can choose whether to auto-generate a value expression or use a custom value expression by using the Source Fetch pop-up menu in the mapping model inspector. If you auto-generate the value expression, supply a key path and a mapping name. The key path is one of the relationships in the Property Mappings section. The mapping name is one of the mappings in the Entity Mappings section.

If you use a custom value expression, enter the expression in the Value Expression text field. Value expressions can be as complicated as you want to make them, but you should stick to simpler value expressions in the mapping model. If you need a complicated value expression to migrate a relationship, write code to handle the migration.

Creating a User Dictionary

The User Info section in the mapping model inspector lets you create a dictionary of key/value pairs for an entity, attribute, or relationship mapping. The information dictionary lets views and controllers access the information contained in the mapping.

Migrating the Data

After getting the mapping model set, you must write the code to perform the migration. Core Data migration is too large a topic to cover completely here. In this section I explain what you have to do to perform automatic migration. For more information on Core Data migration, read the *Core Data Model Versioning and Data Migration Programming Guide*, which is part of Apple's documentation.

Enabling Automatic Migration

The first step you must take to migrate your data is to tell Core Data to automatically migrate its persistent stores. Create a variable of type `NSMutableDictionary`. Set the key `NSMigratePersistentStoresAutomaticallyOption` to YES.

```
NSMutableDictionary* migrationOptions;  
[migrationOptions setObject:[NSNumber numberWithInt:YES]  
    forKey:NSMigratePersistentStoresAutomaticallyOption];
```

Migrating a Document-Based Application

For a document-based application, you must override `NSPersistentDocument`'s `configurePersistentStoreCoordinatorForURL:` method to migrate the data. Pass the `NSMutableDictionary` variable you created when enabling automatic migration as the `storeOptions` argument.

```
NSMutableDictionary* migrationOptions;
NSURL* url;
NSString* fileType;
NSString* configuration;
NSError* error;
BOOL result = [super configurePersistentStoreCoordinatorFor
               URL:url
               ofType:fileType
               modelConfiguration:configuration
               storeOptions:migrationOptions
               error:&error];
```

Migrating a Regular Application

When migrating an application that is not document-based, call `NSPersistentStoreCoordinator`'s `addPersistentStoreWithType:` method. Pass the `NSMutableDictionary` variable you created when enabling automatic migration as the `options` argument. The following code demonstrates migrating an SQLite store:

```
NSMutableDictionary* migrationOptions;
NSPersistentStoreCoordinator* newStoreCoordinator;
NSURL* url;
NSError* error;
NSPersistentStore* newStore;
newStore = [newStoreCoordinator addPersistentStoreWithType:
               NSSQLiteStoreType
               configuration:nil
               URL:url
               options:migrationOptions
               error:&error];
```


Chapter 6

Building Projects

After reading the first five chapters, you know enough about Xcode to create your application. Now you need to take what you've created and build your project into a working application. But there are many decisions you must make when building your project. What compiler should you use? What versions of Mac OS X and iOS should be able to run your application? If you have an iOS application, should it run on iPhones, iPads, or both? If you have a Mac application, should it be sandboxed? If so, what should your application be able to access on the user's Mac? Xcode has many settings that let you control how it builds your projects. In this chapter you will learn how to build your projects and build them the way you want them built.

Project Editor

The project editor is where you modify project and target settings. Select the project file from the project navigator to open the project editor.

On the left side of the project editor are two headings: Project and Targets. Select the project to view and modify project settings. Select a target from the targets list to view and modify target settings. At the top of the editor are buttons to access groups of settings. At the bottom of the editor are buttons to perform tasks such as adding a target, modernizing a project, and adding a build phase to a target.

Targets

Xcode projects consist of one or more targets. A *target* is a set of instructions to build a final product from the files in your project. Common final products are applications, frameworks, and libraries. In this section you'll learn about the following target-related topics:

- Inspecting and modifying target settings.
- Adding targets to your project.
- The types of targets you can add to your project.
- The build phases that make up building a target.

Inspecting and Configuring Target Settings

Xcode targets have lots of settings for you to configure. To inspect and edit a target's settings, select the target from the target list on the left side of the project editor. The project editor has the following groups of target settings:

- Summary
- Info
- Build Settings
- Build Phases
- Build Rules

Only Cocoa and iOS applications have a Summary group. External targets do not have a Build Rules group. Command-line applications do not have an Info group.

Summary

The Summary group shows the most commonly accessed target settings for Mac and iOS application projects.

Mac Target Summary

The Summary group of a Mac application shows the following settings:

- App Icon, which is the application's icon. If you do not supply an icon, your application uses the generic application icon.
- Application Category, which is the category the application should appear under in the Mac App Store. You can ignore this field if you don't plan on submitting your application to the App Store.
- Bundle Identifier, which usually takes the form `com.CompanyName.AppName`.
- Version, which lets you set a version number for the application.
- Build, which lets you set a build number for the application.
- Deployment Target, which is the earliest version of Mac OS X that can run your application. The "Deployment Target" section later in this chapter has more information on deployment targets.
- Main Interface, which is the main xib file. In most cases the main xib file is `MainMenu`.
- A Code Sign Application checkbox. Selecting the checkbox tells Xcode to code sign your application, which you must do to submit the application to the Mac App Store.

- Linked Frameworks and Libraries, which contains a list of frameworks and libraries that link to the application. Click the + button to link a framework or library to the target.
- Entitlements.

Entitlements work with sandboxing and iCloud. Sandboxing limits what an application can access on the user's Mac, which also limits the damage your application can cause if it is compromised. You must sandbox your application to submit it to the Mac App Store. Supporting iCloud allows your application to save files to the user's iCloud account and open them from iCloud.

Select the Entitlements checkbox to turn on entitlements. Selecting the Entitlements checkbox adds an entitlements file to your project. The added entitlements file becomes the choice in the combo box, but you can choose another entitlements file if you want. The entitlements file is a property list file. You specify what your app can access on the user's Mac in the entitlements file.

Select the Enable App Sandboxing checkbox to turn on sandboxing. There are menus that control access to the file system, music folder, movies folder, and pictures folder. Xcode 4.4 adds a menu to control access to the user's Downloads folder. The access levels are no access, read-only access, and read/write access. There are also checkboxes to allow incoming and outgoing network connections, allow access to hardware, and allow access to application data like the user's address book and calendar.

Select the Enable iCloud checkbox to add support for iCloud. If your application stores data using key-value data storage, enter the name of the store in the iCloud Key-Value Store text field. The store is the file that stores the key-value data. Key-value data storage is used to store small amounts of data, such as application state information. An iCloud key-value data store can be no larger than 64 KB. Do not use key-value data storage to store documents.

The iCloud (Ubiquity) Containers table shows a list of container directories associated with your application. Your application saves user documents in a container directory. Click the + button to add a container. An iCloud container generally takes the following form:

`com.CompanyName.AppName`

iOS Target Summary

The Summary group of an iOS application has the following sections:

- iOS Application Target
- iPhone Deployment Info
- iPad Deployment Info
- Linked Frameworks and Libraries
- Entitlements
- Maps Integration

iOS Application Target

The iOS Application Target section has the following settings:

- Bundle Identifier, which usually takes the form `com.CompanyName.AppName`.
- Version, which lets you set a version number for the application.
- Build, which lets you set a build number for the application.
- Devices: iPhone, iPad, or Universal. Use the Devices menu to convert an iPhone application to a universal application.
- Deployment Target, which is the earliest version of iOS that can run your application. The “Deployment Target” section later in this chapter has more information on deployment targets.

iPhone Deployment Info

You must have an iPhone or universal target to have an iPhone Deployment Info section. The iPhone Deployment Info section has the following settings:

- Main Storyboard, which is the name of the application’s main storyboard file. If your application uses xib files instead of storyboard files, leave the Main Storyboard field blank.
- Main Interface, which is the main xib file.
- Supported Interface Orientations: Portrait, Upside Down, Landscape Left, and Landscape Right. Click the button for an orientation to add it to the supported orientations list.
- Status Bar, which lets you control the appearance of the status bar.
- App Icons, one for retina displays and one for devices without retina displays.
- Launch images, one for retina displays and one for devices without retina displays. The launch image is the image that temporarily appears on the screen while the application launches.

iPad Deployment Info

You must have an iPad or universal target to have an iPad Deployment Info section. The iPad Deployment Info section has the following settings:

- Main Storyboard, which is the name of the application's main storyboard file. If your application uses xib files instead of storyboard files, leave the Main Storyboard field blank.
- Main Interface, which is the main xib file.
- Status Bar Visibility. Select the Hide during application launch checkbox to hide the status bar.
- Supported Interface Orientations: Portrait, Upside Down, Landscape Left, and Landscape Right. Click the button for an orientation to add it to the supported orientations list.
- App Icons, one for retina displays and one for devices without retina displays.
- Launch images, one for retina displays and one for devices without retina displays. The launch image is the image that temporarily appears on the screen while the application launches.

Linked Frameworks and Libraries

The Linked Frameworks and Libraries section contains a list of frameworks and libraries that link to the application. Click the + button to add a framework or library to the target.

Entitlements

Select the Entitlements checkbox to turn on entitlements. Selecting the Entitlements checkbox adds an entitlements file to your project. iOS applications use entitlements files for keychain data sharing and iCloud support. Keychain access groups enable keychain data sharing. Click the + button at the bottom of the Keychain Groups table to add a keychain access group.

Supporting iCloud allows your application to save files to the user's iCloud account and open them from iCloud. Select the Enable iCloud checkbox to add support for iCloud. If your application stores data using key-value data storage, enter the name of the store in the iCloud Key-Value Store text field. The store is the file that stores the key-value data. Key-value data storage is used to store small amounts of data, such as application state information. An iCloud key-value data store can be no larger than 64 KB. Do not use key-value data storage to store documents.

The iCloud (Ubiquity) Containers table shows a list of container directories associated with your application. Your application saves user documents in a container directory. Click the + button to add a container. An iCloud container generally takes the following form:

`com.CompanyName.AppName`

The Passes section works with Passbook, which was introduced in iOS 6. To use Passbook in your application, you must get a pass type identifier from Apple. Go to the iOS Provisioning Portal at Apple's iOS Dev Center to request a pass type identifier. Initially Xcode is set to use the pass type identifiers from your provisioning profile. Select the Use selected pass type identifiers radio button if you need more control over the pass type identifiers for the target.

Maps Integration

The Maps Integration section works with the Maps service introduced in iOS 6. Select the Accept transit routing requests checkbox to enable directions. Select the checkbox for each mode of transportation you want to generate directions.

Version and Build Numbers

Most Mac applications have an About box that shows the version number. Some applications also show the build number in the About box. Choose Xcode > About Xcode to open Xcode's About box, which shows the version number and build number. The Summary group allows you to specify a version and build number for the target.

Many developers like to automatically increment the build number every time they build a project. If you want to automatically increment the build number, do not specify a build number in the Summary group. Either write a script to increment the build number or use Apple's `agvtool`. Xcode's Versioning build settings collection contains build settings that work with `agvtool`. Read the "Versioning" section later in this chapter for more information on the Versioning build settings collection.

Adding an Icon to Your Application

To add an icon to your application, you must add the icon file to your project. After adding the icon file to your project, drag it to the App Icon image well in the project editor.

For iOS applications the icon file is a PNG file. You must supply at least two PNG files, one for retina displays and one for non-retina displays, with the retina display image being twice the width and twice the height of the other image. If you have a universal application, you must supply four PNG files: two for iPhone and two for iPad. Drag the PNG files to the appropriate image well.

Mac applications have two different icon creation methods. The new method is to create a folder with the extension `.iconset` and add a set of PNG files to the folder. The old method is to create an ICNS icon file.

Apple's Icon Composer tool can help you create icon files out of PNG files. If you are running Xcode 4.3 or later, choose Xcode > Open Developer Tool > More Developer Tools to access Apple's developer downloads site. Icon Composer is part of the Graphics Tools for Xcode package. Download the package to install Icon Composer. Those of you using an older version of Xcode don't have to worry about downloading Icon Composer because it is installed with Xcode. You can find it in the following location:

`/Developer/Applications/Utilities`

In Icon Composer you will see image wells with labels 512, 256, 128, 32, and 16. Drag a PNG image to each image well. If you have a large PNG file (512-by-512 pixels or larger), you can drag it to each image well, and Icon Composer will scale the image down for each resolution. Saving the document in Icon Composer creates an ICNS icon file. Choosing File > Export To Icon Set creates an icon set. The Icon Composer version that comes with older versions of Xcode cannot export to an icon set.

Unfortunately when you create an icon set in Icon Composer, it does not create high-resolution files for retina displays. If you want to support retina displays, you must manually create the high-resolution files. If you look in the `.iconset` folder Icon Composer creates, you will see the following image files:

```
icon_16x16.png
icon_32x32.png
icon_128x128.png
icon_256x256.png
icon_512x512.png
```

To support retina displays, you must create the following image files and add them to the `.iconset` folder:

```
icon_16x16@2x.png
icon_32x32@2x.png
icon_128x128@2x.png
icon_256x256@2x.png
icon_512x512@2x.png
```

The @2x signifies the icon files are high-resolution. The high-resolution files must be twice the width and twice the height of the other files. The 512-by-512 high-resolution icon must be 1024 pixels wide and 1024 pixels high. After creating the @2x files, add the `.iconset` folder to the project. Open the project editor and drag the `.iconset` folder to the App Icon image well.

Info

The Info group lets you build your target's property list file. This group has the following sections:

- Custom Target Properties
- Document Types
- Exported UTIs
- Imported UTIs
- Services (Mac only)

Custom Target Properties

The Custom Target Properties section lets you configure the target's `Info.plist` property list file. Only projects that use property list files, such as Mac and iOS applications, have a Custom Target Properties section. You can also modify the property list file by selecting it from the project navigator.

When you create a project or add a target to the project, Xcode creates a set of custom properties for the target. Double-click the Value column for a property to change its value. Mac and iOS applications have a Summary group of settings that let you set some custom properties, such as the deployment target, version, identifier, and main xib file. Use the Summary group to set those properties.

If you need to add properties to the property list file, select a key and click the + button in the Key column. A combo box opens. Either choose a property from the combo box list or enter a property name. If you choose a key from the list, the key's data type should be set for you. If you create your own custom property, select a data type from the Type pop-up cell. Enter a value for the key to finish creating the property.

When you look at the values of some properties, you may see values that start with a \$ character, such as `${EXECUTABLE_NAME}` and `${MACOSX_DEPLOYMENT_TARGET}`. These values are the values of Xcode build settings. In most cases you should not directly modify these values in the property list file. Use other sections of the project editor, such as the build settings editor, to change them. For example, set the name of your application by setting the Product Name build setting instead of changing the Bundle name key in the property list.

Document Types

The Document Types section lists the file types your program supports, the file formats it can read and write. Table 6.1 lists the information you can set for document types. You will have one document type for each file format your application can read or write. If your program doesn't read or write files, you can ignore the Document Types section.

To add a document type, click the disclosure triangle next to Document Types to show the list of document types. Click the + button below the document type list to add a document type. To remove a document type, click the Close button (it looks like an X) on the right side of the document type.

In addition to the fields listed in Table 6.1, there is a checkbox named Document is distributed as a bundle. Selecting the checkbox saves the document as a file package. Save the document as a file package when you want to save your application's document data in multiple files and have the data appear as a single file to the user. The developer of a game level editor may want to save the level layout, the level's enemy list, and the level's treasure list in separate files.

Below the fields listed in Table 6.1 is the Additional document type properties section. Use the Additional document type properties section to enter additional information for the document type. The additional properties consist of key-value pairs. If you want to create a Core Data persistent store type for your document, add the following property:

```
Key: Core Data persistent store type
Type: String
Value: Name of the store (built-in store names are Binary,
      SQLite, and XML)
```

iOS Document Types

iOS exposes less of the file system than Mac OS X so there are fewer options for adding document types in iOS applications. Enter the name of your document type in the Name text field. Enter the UTIs for the document type in the Types text field. A custom document UTI takes the form `com.CompanyName.DocumentType`. Click the + button in the icon list to add a document icon.

Table 6.1 Document Type Fields

Field	Description
Name	The name you want to give to the document type. A Quake level editor might have the document name Quake Level.
Class	The subclass of Cocoa's <code>NSDocument</code> class. Only Cocoa applications have to worry about the Class field.
Extensions	The file extension for the document type. Don't place a period before the extension. Adobe Acrobat would have a document type with extension <code>pdf</code> .
Icon	If you create an icon for document files, select it from the combo box or type the file name. Documents without an icon file will have the default Apple document icon.
Identifier	A uniform type identifier (UTI) for the document. If your application has a custom document type, the identifier should take the form <code>com.CompanyName.DocumentType</code> .
Role	Documents can have three possible roles: Editor, Viewer, or None. Editors can view, edit, and save documents. Viewers can view documents, but can't edit or save them. Documents with the None role can't view, edit, or save.
Mime Types	List of MIME (Multimedia Email Extensions) file types, file types a web browser uses. A PDF file has the MIME type <code>application/pdf</code> , and a JPEG file has the MIME type <code>image/jpeg</code> .

Exported and Imported UTIs

A UTI (Uniform Type Identifier) is a string that uniquely identifies an abstract type, such as a file format. A target can have exported and imported UTIs. If your application has a proprietary file format that you want other applications to be able to access, create an exported UTI for your file format. Imported UTIs are for file formats your application did not create, but would like to be able to read. Plug-ins and applications that read proprietary file formats are programs that would create imported UTIs. By adding the imported UTI, someone running your application can open a file that has a proprietary file format without having to install the application that created the file format.

To add exported and imported UTIs, click the disclosure triangle next to Exported UTIs or Imported UTIs to show the list of UTIs. Click the + button below the UTI list to add a UTI. Table 6.2 shows the fields exported and imported UTIs can have.

Table 6.2 Exported and Imported UTI Fields

Field	Description
Description	The name of the UTI.
Identifier	An identifier for the UTI. Your custom UTI's identifier should take the form <code>com.CompanyName.DocumentType</code> .
Icon	If you create an icon for the UTI, select it from the combo box or type the file name. UTIs without an icon file will have the default Apple document icon.
Conforms To	A higher-level UTI that this UTI conforms to. Image files like JPEG and PNG conform to <code>public.image</code> .
Reference URL	A URL that provides a reference for the file's standard. Many open file formats have a specification at the w3.org website. The reference URL for these files would be their file specification page.
Extensions	The file extension for the UTI. Don't place a period before the extension.
Mime Types	List of MIME (Multimedia Email Extensions) file types, file types a web browser uses.
Pboard Types	UTI strings for the pasteboard types this format supports. A UTI that can deal with RTF data in the pasteboard would use the string <code>public.rtf</code> .
OS Types	Four-character codes for the file type. Before Mac OS X, Mac applications used OS Types instead of file extensions to identify themselves as the creator of a particular file.

Below the fields in Table 6.2 is the Additional UTI properties section. The additional UTI properties are a collection of key-value pairs. Click inside the table view to add a UTI property.

URL Types

The URL Types section allows you to create a custom URL scheme for your application. A custom URL scheme is a URL protocol handler, which is the first part of a URL. The most familiar example of a URL protocol is `http://`. By creating a custom URL scheme, you can create a URL specific to your application. When someone clicks the URL link, your application launches.

To add a URL type to a target, click the disclosure triangle next to URL Types to show the list of URL types. Click the + button below the URL type list to add a URL type. The URL Type section has the following fields:

- Identifier, which uniquely identifies the URL type. It should take the form `com.CompanyName.AppName`.
- Icon, which is the icon file for the URL type.
- URL Schemes, which contain the contents of the scheme. An example of a URL scheme is `appname://`.
- Role, which can be Editor, Viewer, or None.

Below the four fields is the Additional url type properties section. The additional URL type properties are a collection of key-value pairs. Click inside the table view to add a URL type property.

Services

The Services section is for Mac applications that provide services that can be accessed through the Services menu. Choose `AppName > Services` to access the Services menu. If you select some text in Xcode, the available services include creating a sticky note, searching Google, and creating an email.

To add a service to a target, click the disclosure triangle next to Services to show the list of services. Click the + button below the service list to add a service. The Services section has the following fields:

- Method Name, which is the name of the service.
- Port Name, which is usually the name of your application.
- Menu Item, which specifies the text of the Services menu item.
- Send Types, which are data types sent from the application requesting the service.
- Return Types, which are data types returned to the application requesting the service.

Below the five fields is the Additional service properties section. The additional service properties are a collection of key-value pairs. Click inside the table view to add a service property.

Build Settings

The Build Settings group is where you configure the build settings for a target. If you select the project on the left side of the project editor, you will notice that the project also has a Build Settings group. Both Build Settings groups have the same settings to configure. What is the difference between the two Build Settings groups?

In Xcode 4 there is not much difference between the two Build Settings groups. To see the difference, click the Levels button at the top of the build settings editor. The build settings editor for a target has a column for the target. This column is missing in the project's build

settings editor. When in doubt, select the target from the project editor when you want to configure build settings. Read the “Xcode Build Settings” section later in this chapter for information on individual build settings.

Target Build Phases

Build phases are steps Xcode takes to build a target. Table 6.3 lists the possible build phases a target can have. Each target type has its own set of build phases. A Cocoa application target initially has four build phases: Target Dependencies, Compile Sources, Link Binary with Libraries, and Copy Bundle Resources.

Clicking the disclosure triangle next to a build phase shows the files that are in the build phase. Click the + button to add a file to a build phase. Select a file from the list and click the minus button to remove the file from a build phase.

Target Dependencies

A *target dependency* is a target Xcode must build before it builds the current target. Your project must have multiple targets before you can set target dependencies. Aggregate targets require target dependencies. The target dependencies are the targets that make up the aggregate target.

Another common case where a target uses target dependencies is when a unit testing target for an application project requires the application to be built before running the tests. The application target is a target dependency of the unit testing target.

To add a target dependency to a target, click the Target Dependencies disclosure triangle and click the + button. A sheet opens that has a list of your project’s targets. Select a target from the list and click the Add button to create the dependency.

Table 6.3 Target Build Phases

Build Phase	Description
Target Dependencies	Targets Xcode must build before it builds the current target.
Compile Sources	Compiles source code files. You can add compiler flags for a single file in this build phase.
Link Binary with Libraries	Links the object files Xcode creates during compilation to the frameworks and libraries you added to your project.
Copy Bundle Resources	Copies files that support your source code files, such as xib files, property list files, audio files, and image files, to the proper location in the final product.
Copy Headers	Copies header files to the proper location in the final product. If you're creating a framework, you need the Copy Headers build phase.
Run Scripts	Runs shell scripts when building the project.
Copy Files	Copies files from the project directory to a location you specify.
Build Carbon Resources	Compiles resource files that contain Carbon Resource Manager resources. You must add a Rez resource file to your project and give the file the extension <code>.r</code> in order to add the Build Carbon Resources build phase to your target. Most of you will never use this build phase.

Adding Build Phases

The build phases Xcode includes for a target depend on the target. If you need a build phase that Xcode didn't supply, you can add a build phase by clicking the Add Build Phase button. Choose the build phase you want to add from the menu that opens when you click the Add Build Phase button.

The Copy Files and Run Scripts build phases are the ones you're most likely to add. Use the Copy Files build phase when your program needs a file, framework, or library to be in a specific location. Use the Run Scripts build phase if you need to run a script when building your project. If your program includes files written in languages that Xcode doesn't directly support, write a script to compile the files that Xcode can't compile for you.

Reordering Build Phases

When Xcode builds your project, it goes through the build phases in the order shown in the project editor. The default order works in most cases, but if you need to change the order, select a build phase and drag it to where you want it to appear in the build order.

Be careful when ordering build phases. If you place build phases in the wrong order, Xcode won't be able to build your project correctly. Placing the Link Binary with Libraries build phase before the Compile Sources build phase will cause problems.

Build Rules

The Build Rules group lets you specify the programs Xcode should use to process files. A lot of the rules are no-brainers; Xcode has rules to compile data model files with the data model compiler, Interface Builder files with the Interface Builder compiler, and DTrace source files with DTrace. Most of you won't have to deal with build rules. The most common case to add build rules is when you need to run a custom script on some files in your project. Create a rule that tells Xcode to use your script to process the appropriate files.

There are two ways to add build rules. First, you can click the Add Build Rule button at the bottom of the project editor to add a build rule. Second, you can click the Copy to Target button next to an existing build rule. Click the Copy to Target button if you want to use a custom script to build a file that has an existing build rule.

When you create a build rule, you will see two pop-up menus: Process and Using. The Process pop-up menu specifies the file the build rule applies to. If your file type does not appear in the menu, choose Source files with names matching. If you want to create a build rule to process Ruby files, you should choose Source files with names matching and enter `*.rb` in the text field.

The Using pop-up menu specifies the tool to use to build the files. Choose Custom script if the tool you want to use does not appear in the menu. Enter your script in the text view below the Using pop-up menu or drag an existing script to the text view. A custom script can create output files. Click the + button to add the name of an output file.

Adding Targets

When you create a project, Xcode supplies one target for the project. You can usually get away with using the target Xcode provides, but there are instances where you may need to create multiple targets. Suppose you're writing a library and you want to write a test application to make sure the code you wrote for the library is correct. In this case you would have two targets: one target to build the library and one target to build the test application.

To add a target to one of your projects, click the Add Target button at the bottom of the project editor. The New Target Assistant window opens. You will notice that the New Target Assistant looks like the New Project Assistant. Creating a target is similar to creating a project, except you don't have to pick a location on your hard disk to save the target.

Most of the target types are identical to project types, but there are two targets that have no corresponding project type: aggregate targets and unit testing bundle targets.

Aggregate Targets

Use an aggregate target to build a group of targets. The aggregate target depends on the targets that make up the aggregate. When you tell Xcode to build the aggregate target, it builds each target in the aggregate target.

Unit Testing Bundles

Unit testing involves writing and running automated tests on your program's code. Unit testing is a big part of test-driven development, which is part of the agile software development process. Xcode has unit testing bundle targets for Cocoa and iOS programs, which simplifies unit testing for Objective-C programmers. Adding the unit testing bundle target is the first step to unit testing. There are several more steps to take.

1. Add a target dependency for the unit test target.
2. Configure the unit test bundle.
3. Add unit testing classes.
4. Write the test cases.
5. Run the tests.

If you tell Xcode to create a unit testing target when creating the project, Xcode should add a target dependency and configure the unit test bundle for you.

Adding a Target Dependency

Adding a target dependency is not a mandatory step, but it allows you to run your unit tests automatically. To be able to unit test your application, Xcode must build your application before building the unit test target. The target dependency ensures the application builds when you build the unit test target. In this scenario your application is a target dependency of the unit test target. To add a target dependency, perform the following steps:

1. Select the project from the project navigator.
2. Select the unit testing target from the left side of the project editor.
3. Select Build Phases at the top of the project editor.
4. Click the disclosure triangle next to Target Dependencies.
5. Click the + button to add a target dependency.
6. A sheet with a list of your project's targets opens. Select a target from the list and click the Add button.

Configuring the Unit Test Bundle

You must configure the unit test bundle if you're unit testing an application. To configure the unit test bundle, modify the Bundle Loader and Test Host build settings. You must set the value of these two build settings to the path to the application's executable file.

```
$(BUILT_PRODUCTS_DIR)/MyApp.app/Contents/MacOS/MyApp (Mac)
$(BUILT_PRODUCTS_DIR)/MyApp.app/MyApp (iOS)
```

You can avoid entering the executable path twice. Enter it for the Bundle Loader build setting, and give the Test Host build setting the following value:

```
$(BUNDLE_LOADER)
```

NOTE

iOS applications running in the simulator should leave the Test Host build setting blank. The simulator does not support application-hosted unit tests. If you set the Test Host build setting, you will get a warning when building the unit testing target and the tests will not run.

Adding Unit Testing Classes

Choose File > New > File to add a unit testing class to your project. The Cocoa unit testing class is in the Cocoa group. The iOS unit testing class is in the Cocoa Touch group. When you add a unit testing class, you must tell Xcode what targets the class should be added to. The unit testing class should be added to the unit testing target only. Xcode initially does the wrong thing with unit testing classes. It sets the class to be added to the application (or whatever the non-unit testing target is) target, not the unit testing target. Make sure the unit testing class is added only to the unit testing target.

If you're unit testing a 64-bit Mac application, you might get linker errors when building the unit testing target. You can get linker errors if the class files being tested are not members of the unit testing target. If you get linker errors when unit testing a 64-bit Mac application,

add the class files being tested, the implementation files in your application, to the unit testing target. Use the file inspector to make a file a member of the unit testing target. The section “File Inspector” in Chapter 1, “Xcode Projects”, has additional information on the file inspector.

Writing and Running Unit Tests

After adding unit testing classes, write the test cases that test your program’s code. If you practice test-driven development, you write a test, write code to get the test to pass, write another test, get that test to pass, and so on. Writing test cases is beyond the scope of this book.

When you finish writing a test case, run the tests. Xcode can run your unit tests after building the unit test target by choosing Product > Build For > Testing, or you can run the tests yourself by choosing Product > Test. If you want Xcode to run your tests after building the unit test target, you must set the Test After Build build setting to Yes. Unit testing goes smoother for me when Xcode runs the tests after building the unit test target.

The results of the tests appear in the build results window if you tell Xcode to run the tests after building the unit test target. A failing test appears as a build error. If the Test After Build build setting is set to No, the tests appear in the debug console and in the log navigator.

Project Settings

Select the project from the list on the left side of the project editor to view and modify project settings. There are two buttons at the top of the project editor: Info and Build Settings. When you click the Info button, you will see the following sections in the editor:

- Deployment Target
- Configurations
- Localizations

The Build Settings group is where you configure the project’s build settings. Refer to the “Xcode Build Settings” section later in this chapter for more detailed information on build settings.

Deployment Target

The *deployment target* is the earliest version of Mac OS X or iOS that can run your program. When you create an Xcode project, Xcode sets the deployment target to the version of Mac OS X you're running or the version of the iOS SDK you installed. If you're writing a program for personal use, you can avoid changing the deployment target. But if other people are going to use your program, you should change the deployment target unless you're writing a cutting-edge application that uses the latest features in Mac OS X or iOS.

Changing the deployment target lets your program run on older versions of Mac OS X or iOS, assuming your code uses APIs that work on the deployment target. If you use Scene Kit and set the deployment target to Mac OS X 10.7, your code isn't going to run on 10.7 because Scene Kit was introduced in Mac OS X 10.8. If you use a method introduced in iOS 5, your application won't run on a device running iOS 4.

What Should My Deployment Target Be?

You should set the deployment target to the earliest version of Mac OS X or iOS you can reasonably support. For Mac applications a general rule is to support the latest version of Mac OS X plus the previous 1-2 versions. Because Xcode 4 dropped support for PowerPC Macs, there is little reason to support anything earlier than Mac OS X 10.6 in an Xcode 4 project. The main reason to support Mac OS X 10.5 and earlier is to allow your application to run on PowerPC Macs. Install Xcode 3.2 if you need to support PowerPC Macs.

For iOS applications the deployment target depends on the version of Xcode you're using and the devices you want to support. If you are using Xcode 4.5 or later, iOS 5.0 is a reasonable deployment target. The earliest deployment target you can set is iOS 4.3, and every device that runs iOS 4.3 can also run iOS 5.0.

If you are running a version of Xcode that can set earlier deployment targets, the deployment target depends on the devices you want to support. Setting the deployment target to iOS 3.1 allows your application to run on all iOS devices. Setting the deployment target to 4.2 allows your application to run on all devices except the original iPhone and the first-generation iPod Touch.

Remember that I said you should set the deployment target to the earliest operating system version you can reasonably support. If an Apple technology shaves months off your development time or adds something essential to your application, don't be afraid to use it and limit your application to people running newer versions of Mac OS X and iOS.

Deployment Targets and SDKs

An SDK contains everything you need to build an application for a particular version of Mac OS X or iOS. The SDK for a particular version of Mac OS X or iOS contains the latest technologies your application can use. When you install the iOS SDK, it overwrites any previously installed versions of the SDK. iOS projects are limited to using the SDK you installed.

Mac Xcode projects initially use the latest SDK installed on your Mac. In most cases there is no need to change SDKs. The SDK has no effect on what versions of Mac OS X can run your program. The deployment target determines the versions of Mac OS X that can run your application.

Normally the best course of action is to use the latest SDK while using the earliest deployment target you want to support. By using the latest SDK your application can take advantage of any bug fixes Apple made.

When Should You Use an Earlier SDK?

Last section I said there was rarely a need to change SDKs. When would you want to use an earlier SDK? I can think of three reasons why you might want to use an earlier SDK. The first reason is if you want to build a version of your application for a specific version of Mac OS X. Suppose you have two versions of your application: one for Lion and one for Mountain Lion. You might use the 10.7 SDK for the Lion version and the 10.8 SDK for the Mountain Lion version.

The second reason is for testing purposes. Suppose you're supporting Lion and Mountain Lion, and you want to make sure you didn't use any Mountain Lion-specific function calls. You would temporarily use the 10.7 SDK so you could discover any compatibility errors when you build the project. When your code compiles without any problems with the 10.7 SDK, switch back to the 10.8 SDK. The Base SDK build setting lets you temporarily change the SDK. You could use the 10.7 SDK for the Debug build configuration and the 10.8 SDK for the Release build configuration.

The third reason to use an older SDK is if you have some old code. A newer SDK may force you to update some of the old code to get it to build. You may not want to spend the time to update your code so you would use an older SDK.

Build Configurations

A target is a set of instructions to build a final product from the files in your project. The way you want to build the final product changes as you get closer to finishing your program. When you start writing your program, you want to turn on debugging symbols and turn off code optimization so you can examine your program and correct any mistakes you made. When you have your code ready for release, you want to remove debugging symbols from the executable file to reduce the size of the final product and turn on code optimization to make your code run faster.

Build configurations solve the problem. They allow you to build the same target in different ways, reducing the need for you to create new targets. Targets tell Xcode what to build. Build configurations tell Xcode how to build the target.

Xcode supplies two build configurations for each project: Debug and Release. The Debug build configuration contains settings that make debugging easier. The Release build configuration contains settings to build a release version of your program. To look at your project's build configurations, select the project from the project navigator and click the Info button at the top of the project editor.

To add a build configuration to the list, click the + button below the list of build configurations. You have the choice of duplicating any of the existing build configurations. Enter the name you want to give the build configuration. The Debug and Release build configurations Xcode supplies should be sufficient for most of you, but the following are examples of build configurations you may want to add:

- A static analysis build configuration that runs the static analyzer when building the project.
- Build configurations for different code optimization levels.

To delete a build configuration, select it from the list and click the minus button.

If you prefer to build your projects from the command line instead of from Xcode, the Command-line builds use pop-up menu lets you specify the build configuration to use when building from the command line.

When looking at the table of build configurations, you noticed a Based On Configuration File column. A configuration settings file is a text file that contains build settings. Configuration settings files keep you from having to modify the same build settings in multiple projects. You must add a configuration settings file to your project before you can base a build configuration on a configuration settings file. Read the “Configuration Settings Files” section later in this chapter for more information on using configuration settings files.

Localizations

The Localizations section contains a list of spoken languages your application supports. When you create a project, Xcode creates one set of localized files for the spoken language you're using, which is most likely English. For each language your application supports, Xcode tells you the number of localized files in the project.

If you want to support other spoken languages in your application, click the + button. Choose a language from the menu. A sheet opens with a list of files to localize. If you don't want to localize a file, deselect the checkbox next to it. Click the Finish button. When you click the Finish button, Xcode adds a set of localized files to your project. The files Xcode adds depend on the project, but examples of localized files are xib files, `InfoPlist.strings`, and `Credits.rtf`.

Xcode 4.4 adds a Use Base Localization checkbox for Cocoa and iOS application projects. You must be using the iOS 6 SDK to be able to do anything with the checkbox in an iOS project. Selecting the checkbox creates a new base localization that is used to localize other files. The base localization helps when localizing xib files. By using a base localization Xcode uses one xib file with a strings file for each spoken language your application supports. Separating the localizable data from the xib file simplifies translation.

When you select the Use Base Localization checkbox, a sheet opens with a list of files to localize, which most likely is a list of xib files in your project. If you don't want to localize a file, deselect the checkbox next to it. Click the Finish button to complete the localization.

After clicking the Finish button, you will see a new entry in the Localizations list named Base. If you add a new localization, you will see Base in the Reference Language column. In the File Types column you will see a Localization String entry for each xib file. If you click the Finish button and look at a xib file in the project navigator, you will see a strings file for the new language.

Xcode Build Settings

Click the Build Settings button at the top of the project editor to access Xcode's build settings. There are two groups of buttons above the list of build settings: Basic/All and Combined/Levels. The Basic/All group controls what build settings appear in the editor. Clicking the Basic button shows the most commonly accessed build settings, and clicking the All button shows every build setting. Use the search field to filter the contents of the build settings editor. You can search on the following criteria: Name, Title, Definition, Value, Category, Description, or all of them. Click the magnifying glass icon in the search field to choose the search criteria.

The Combined/Levels group controls the number of columns that appear in the build settings editor. Clicking the Combined button shows one column of build settings. Clicking the Levels button shows a column of settings for each build settings level: Default, Project, Target, and Resolved. You must select a target from the project editor's target list to see the Target column. Showing the levels allows you to see any differences in project and target build settings as well as any differences from the default values. Showing combined build settings makes setting a build setting's value easier because you don't have to worry about setting the value for the project or target.

Build settings have two types of controls. The type of control depends on the setting. Settings that can have one of several possible values have a pop-up cell. Make your choice from the cell's menu. The rest of the settings require you to enter text. To change the settings that require you to enter text, select the setting and double-click the appropriate column in the editor. A pop-up editor opens for you to enter the setting's value.

If you're a beginning programmer, the number of settings can be intimidating, but don't freak out. Xcode configures the settings in a way that works well in general cases. You can stick with Xcode's default settings initially. If you find that Xcode isn't building your project the way you want, you can go back and configure the build settings. Depending on your project's requirements, you'll end up not touching 80-99% of the build settings.

Another thing you will notice when looking at the build settings is that some settings are blank. This is fine. You don't need to fill every build setting.

Due to the number of build settings Xcode provides, I can't provide a detailed explanation of every setting. I'm going to focus on the build settings you're most likely to modify. If you're interested in a build setting I don't explain in the book, open the Quick Help inspector by choosing View > Utilities > Show Quick Help Inspector. When you select a build setting, Quick Help displays information about that setting.

Architectures

The Architectures collection specifies the processor architectures to build for and the SDKs used to build the project. Xcode supports the following architectures: Intel and ARM. The Intel architecture is used when building Mac projects, and the ARM architecture is used when building iOS projects.

For Mac OS X projects, Xcode 4's standard configuration is to build 32 and 64-bit Intel versions, but projects created in newer versions of Xcode are configured to build for 64-bit Intel only. On iOS projects, Xcode is set to build for the appropriate architecture, which depends on the project you create. Xcode's initial configuration works well in most cases.

The main reason you would have to mess with the Architectures build setting for a Mac project is if you wanted to add or remove support for 32-bit Intel. Use the menu to change the architectures. There are the following options for Mac projects:

- 32-bit Intel, which builds a 32-bit version for Intel only.
- 64-bit Intel, which builds a 64-bit version for Intel only.
- Native Architecture of Build Machine, which builds a version for your Mac's architecture. Since Xcode 4 runs only on Intel Macs, there is little point in building for the native architecture.
- Other, which lets you manually specify the architecture.

There are two architecture options for iOS.

- Standard, which builds for `armv7` and `armv7s` in Xcode 4.5 and later and builds for `armv7` in Xcode 4.2, 4.3, and 4.4.
- Other, which lets you manually specify the architecture. Choose Other if you need to add `armv6` support in Xcode 4.2, 4.3, or 4.4.

Remember that the more architectures you build for, the longer Xcode takes to build your project. If you're building 32/64-bit Intel, Xcode has to build two versions of your program: 32-bit Intel and 64-bit Intel. Building two versions takes longer than building one. Build only for the architectures you need.

The Valid Architectures setting contains a list of architectures you can build for. Xcode provides the following valid architectures:

- `i386` is 32-bit Intel.
- `x86_64` is 64-bit Intel.
- `armv6` is the architecture for the original iPhone and iPhone 3G.
- `armv7` is the architecture for the iPhone 3GS, iPhone 4, iPhone 4S, and iPad.
- `armv7s` is the architecture for the iPhone 5.

Do not modify the Valid Architectures build setting. Its purpose is to show you the valid values so you can set the architecture manually.

The Build Active Architecture Only build setting tells Xcode to build only for the active architecture, which is the architecture of your Mac initially. Since Xcode 4 runs only on Intel Macs and does not create PowerPC binaries, there is little need for this build setting.

The Supported Platforms build setting specifies the platforms your project supports. Xcode supports the `macosx`, `iphoneos`, and `iphonesimulator` platforms. You should not have to modify this build setting.

The Base SDK build setting determines the SDK Xcode uses to build your project. You may need to change the Base SDK if you're opening an old Xcode project that uses an earlier SDK than the ones installed on your Mac. Changing the Base SDK can be useful if you're supporting older versions of Mac OS X. Suppose you want to support Mac OS X 10.7 and 10.8. You could set the Base SDK in the Debug version to 10.7 to make sure everything compiles and set the Base SDK in the Release version to 10.8 to take advantage of any bug fixes in the 10.8 SDK. In most cases you should use the newest SDK. iOS projects can use only the installed SDK because iOS SDK installations overwrite previously installed SDKs. Refer to the section "Deployment Targets and SDKs" earlier in this chapter for more detailed information on SDKs.

The Additional SDKs build setting is for sparse SDKs, which are SDKs supplied by third parties or built by you. Sparse SDKs are used more in iOS development because dynamically linked libraries are not allowed on iOS.

Build Locations

When Xcode builds your project, it creates files. The files Xcode creates depend on the project type and the programming languages used. Some files that Xcode creates are object files, libraries, frameworks, and executable files (applications). Xcode initially places these files in the derived data directory, which is initially in the following location:

```
/Users/Username/Library/Developer/Xcode/DerivedData
```

When you build the project, Xcode creates a folder for your project in the derived data folder and places the files it creates in that folder.

The Build Locations collection lets you specify a different location for build products and other files Xcode creates, but you shouldn't need to use this collection often. Xcode's Locations preferences let you specify a different global derived data location. Choosing File > Project Settings lets you specify a new derived data location and build location for a project, reducing the need to use the Build Locations collection. The Build Locations collection lets you specify build locations for a single build configuration.

Build Options

The Build Options collection contains miscellaneous build options. Use the Build Options collection to pick a compiler to use to build your project. Read the next section to learn more about choosing a compiler.

Use the Debug Information Format setting to specify your project's debug information format. There are two options: DWARF and DWARF with dSYM File. If you see a Stabs option, don't use it. Stabs is deprecated. Release builds should use DWARF with dSYM because the debugging symbols get placed in an external dSYM file. Placing the symbols in an external dSYM file reduces the executable file size, which you want for release builds. Refer to the section "Choosing a Debugging Format" in Chapter 7, "Debugging", for more information on debugging formats.

Activating the Run Static Analyzer build setting tells Xcode to run the Clang static analyzer when it builds your project. You don't have to activate this build setting to run the analyzer. Choosing Product > Analyze will run the static analyzer for you. The point of the Run Static Analyzer build setting is to automatically run the analyzer when building your project.

Setting the Generate Profiling Code setting to Yes tells the compiler to generate profiling code. This build setting should be set to No. Apple's profiling tools do not use the Generate Profiling Code build setting.

Picking a Compiler

Use the Compiler for C/C++/Objective-C build setting in the Build Options collection to pick the compiler to use to build your project. There are two compiler choices: LLVM GCC 4.2 and LLVM.

LLVM GCC 4.2 uses GCC for the front end and LLVM for the back end. The LLVM compiler uses Clang for the front end and LLVM for the back end. Applications compiled with LLVM should run faster than ones compiled with GCC. Compiling with the Clang front end is faster and provides better error messages. In most cases you should use LLVM to compile your code.

What is the difference between the front end and back end? The front end analyzes your source code, which includes checking for syntax errors. The back end optimizes your code and generates machine code.

Code Signing

The Code Signing build settings collection deals with code signing, which lets you provide a signature for your application that identifies you as the author of the code. Code signing provides security benefits if your program is available for people to download. It lets the person downloading the program know the code came from you and it wasn't altered.

There are two things to keep in mind about code signing. First, code signing is meant for applications that are available to the general public. If you're using Xcode to learn programming or to write programs for personal use, don't worry about code signing.

Second, code signing is something you do when your application is ready for release. Don't worry about code signing at the start of development. For those of you writing iOS applications, you don't need code signing until you're ready to move your application to a device for testing. When you start development, you'll be testing your application on the iOS Simulator that comes with the iOS SDK. You can run your application on the simulator without code signing it.

Code Signing iOS Applications

iOS applications require code signing to load them on your device and to submit them to the App Store. To code sign an iOS application, you must join the iOS Developer Program and pay the annual membership fee.

Joining the developer program gets you a signing certificate, an application ID, and a provisioning profile. It also gets you access to information on running your applications on your device and distributing your applications.

The iOS Developer Program has a Development Provisioning Assistant that walks you through the creation of a provisioning profile and signing certificate.

Code Signing Mac Applications

Mac applications require code signing to submit them to the App Store and to be recognized by the Gatekeeper feature introduced in Mac OS X 10.8. To submit applications to the App Store, you must join the Mac Developer Program and pay the annual membership fee. When you join the Mac Developer Program, you get access to the App Store Resource Center, which provides all the information you need to submit your application to the App Store, including code signing.

Gatekeeper is a security feature in Mac OS X. In its initial setup Mac OS X will not allow a user to run downloaded applications from unidentified developers. To identify yourself you must join the paid Mac Developer Program, get a developer ID from Apple, and code sign your application. Getting a developer ID from Apple and code signing make it easier for users to launch your application.

Even if you don't plan on submitting your application to the App Store, code signing can help you. Code signing helps most when updating your applications. If you use the same code signature for both versions, the operating system knows the updated version came from you and grants access to the user's keychain without asking the user to verify.

Creating a Code Signing Identity

To use code signing, you must create a code signing identity, which is a certificate identifying you as the creator of your application. Launch the Keychain Access application to create a certificate. You can find it in your Applications folder under Utilities. Those of you writing Mac applications that will not be submitted to the App Store can choose Keychain Access > Certificate Assistant > Create a Certificate to create a self-signed certificate or choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority to get a certificate from a third party. Third-party certificates are more trusted than self-signed certificates, but third-party certificates cost money.

iOS developers and Mac developers submitting applications to the App Store should choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority. Select the Request is Saved to disk radio button and the Let me specify key pair information checkbox. Click the Continue button and pick a location to save the certificate. Choose 2048 for Key Size and RSA for the algorithm. Click the Continue button to finish creating the code signing identity.

Code Signing Build Settings

After creating a code signing identity, you can use Xcode's code signing build settings. There are four settings.

- Code Signing Entitlements
- Code Signing Identity
- Code Signing Resource Rules Path
- Other Code Signing Flags

The Code Signing Entitlements build setting is the name of the entitlements file. Entitlements files work with iCloud and Mac app sandboxing. Selecting the Entitlements checkbox in the target's Summary section adds an entitlements file to your project. The entitlements file Xcode created is the initial value for the Code Signing Entitlements build setting. Double-click a column in the build settings editor to enter a different entitlements file.

The Code Signing Identity build setting is the name of the certificate you created in Keychain Access. You will get a build error if the certificate is missing, invalid, or its name is misspelled. Choose Don't Code Sign if you don't want to code sign your project.

The Code Signing Resource Rules Path build setting contains the path to a resource rules file. Resource rules files are optional. They contain instructions for copying resources like audio and graphics files to the application bundle when building the application. You would need a resource rules file if you wanted to override the default method of copying resources to

the application bundle. If you need a resource rules file, you can add one to your project by choosing File > New > File. The resource rules file is in the Other section under iOS. Mac applications do not use resource rules files.

Xcode uses the command-line tool `codesign` to create code signatures. The Other Code Signing Flags build setting lets you add flags to `codesign` that aren't covered by the other code signing build settings.

Deployment

The Deployment collection determines how the final product is installed on the user's machine. You can tell Xcode where to install the product, the user that owns the product, the group that owns the product, and the file permissions to install the product files. Settings of particular interest are the deployment target, the Targeted Device Family build setting, and the settings related to stripping symbols.

Deployment Target

The deployment target is the earliest version of Mac OS X or iOS your program will run on. Normally you set the deployment target by clicking the Info button for the project or the Summary button for the target in the project editor. Refer to the "Deployment Target" section earlier in this chapter for more detailed information on deployment targets.

If you choose to set the deployment target from the build settings editor, Mac applications should set the OS X Deployment Target build setting. iOS applications should set the iOS Deployment Target build setting.

Targeted Device Family

The Targeted Device Family build setting is for iOS applications. It determines what devices the application is built for. There are three possible values: iPhone, iPad, and iPhone/iPad. Choose iPhone/iPad if you want to build a universal application. You can also set the devices for a target in its Summary group of settings.

Stripping Symbols

Symbols make up most of the code you write: classes, variables, functions, enumerated data types, constants, and macros. There are also debugging symbols that make it possible to debug your programs. In the Debug build configuration the project and debugging symbols

get copied into the binary file. Copying debug symbols is good because having the symbols makes debugging a lot easier. Imagine how difficult debugging would be if you didn't have the names of your variables and functions.

But when you're ready to release your program, copying the symbols becomes a problem. In a release build you want to strip the symbols out of the binary to make the file size smaller. Setting the Strip Linked Product build setting to Yes strips the linked product (application, library, or framework) of symbols. Setting the Strip Debug Symbols During Copy build setting to Yes strips symbols out of any files that are part of the Copy Files and Copy Bundle Resources build phases. If you find the symbols aren't being stripped, set the Deployment Postprocessing build setting to Yes.

The Strip Style build setting determines what symbols get stripped. The initial value is All Symbols, which means everything gets stripped. You can also use two other styles. Choosing Non-Global Symbols strips all non-global symbols but saves external symbols. Choosing Debugging Symbols strips debugging symbols, but saves local and global symbols. There is not much point to stripping only debugging symbols. Setting the Debug Information Format build setting to DWARF with dSYM accomplishes the same thing; the debugging symbols go into an external file.

Setting the Use Separate Strip build setting to Yes tells Xcode to make a special call to the `strip` tool instead of stripping during linking.

Kernel Module

The Kernel Module collection is used only by kernel extension projects so most of you can ignore this collection. The Kernel Module collection lets you set a kernel module's name, start routine, stop routine, and version.

Linking

The Linking collection contains settings for the linker. When Xcode builds your project, the compiler compiles your source code files into object files. The linker links the object files with frameworks and libraries to create a final product, such as an application or a library.

The two settings you're most likely to use are Other Linker Flags and Dead Code Stripping. If you don't see an equivalent build setting in the Linking collection, add the flag to the Other Linker Flags build setting. Linking a library to your project is a common case where you would add a linker flag. You would use the following flag to link a library:

`-lLibraryName`

Setting the Dead Code Stripping build setting to Yes tells the linker to strip any unused code out of the executable. Dead code stripping is good for release builds because it reduces the size of your executable file.

If you're writing a C++ program and you get linker errors, activating the Display Mangled Names build setting may help you. Because multiple C++ functions can have the same name, the compiler mangles the function names to give each function a unique name. Displaying the mangled names can help locate the source of a link error.

If you're unit testing an application, supply the path to your application's executable file to the Bundle Loader build setting.

```
$(BUILT_PRODUCTS_DIR)/MyApp.app/Contents/MacOS/MyApp (Mac)
$(BUILT_PRODUCTS_DIR)/MyApp.app/MyApp (iOS)
```

Packaging

The Packaging collection provides options on packaging the product Xcode creates when it builds your project. The most interesting setting in this collection is the Product Name build setting. The product name is initially set to be the name of your project. If you want a different name for your application, change the product name. If you're going to change the product name, you should change it in the target. The target's build settings override the project's build settings.

Search Paths

The Search Paths collection is where you tell Xcode to search for header files, libraries, and frameworks. If you limit yourself to Apple's frameworks, you shouldn't need to change search paths.

Programs that use third-party libraries and frameworks are the ones most likely to specify search paths. If you add a third-party library or framework to your project, you may get compiler or linker errors even though there's nothing wrong with the code. The errors may be caused by Xcode being unable to find the libraries and frameworks you added. In this case you'll need to add search paths for the libraries and frameworks you added.

Unit Testing

Unit testing involves testing your program's functions to make sure they're correct. There are many tools available to make unit testing easier. The Unit Testing collection contains settings for using unit testing tools. Refer to the "Unit Testing Bundles" section earlier in this chapter for more information on unit testing.

The Test After Build build setting determines when unit tests are run. If you set Test After Build to Yes, Xcode runs the unit tests after building the unit testing target. If Test After Build is set to No, you must run the tests yourself by choosing Product > Test.

Only application projects use the Test Host setting. The unit testing tool inserts the unit tests in the host. Supply the path to the application's executable file as the Test Host setting's value.

```
$(BUILT_PRODUCTS_DIR)/MyApp.app/Contents/MacOS/MyApp (Mac)
$(BUILT_PRODUCTS_DIR)/MyApp.app/MyApp (iOS)
```

NOTE

iOS applications running in the simulator should leave the Test Host build setting blank. The simulator does not support application-hosted unit tests. If you set the Test Host build setting, you will get a warning when building the unit testing target and the tests will not run.

The Test Rig setting is where you specify the testing tool to use. Supply the path to the unit testing tool as the Test Rig setting's value. You should not have to modify this setting if you're using Objective-C and you use the unit testing bundle and unit testing classes that come with Xcode.

Versioning

The Versioning build settings collection works with Apple's `agvtool`, which is a versioning tool for Xcode projects that automates adding build version numbers for your project. There are two build settings you must set to enable versioning with `agvtool`: Versioning System and Current Project Version. The Versioning System build setting is initially set to None. You must set it to Apple Generic to turn on versioning. After you enable versioning, you must set the Current Project Version build setting. It must be a numeric value. A good starting value would be 1.

Code Generation

The Code Generation collection contains build settings that affect the machine code the compiler generates. Many of the build settings in this collection help improve application performance. Most of you won't have to deal with many of the settings in this collection, but there are two settings you are most likely to use: Optimization Level and Generate Debug Symbols.

Optimization Level

When you create a project, Xcode sets the optimization level to None for the Debug build configuration and sets the optimization level to Fastest, Smallest for the Release build configuration. The initial settings work for most cases.

The Fastest, Smallest optimization level produces the fastest code that doesn't increase code size. Smaller code generally runs faster than larger code. If you need speed more than small code size, change the optimization level for the release build from Fastest, Smallest to Fastest.

Generate Debug Symbols

Setting the Generate Debug Symbols build setting to Yes tells the compiler to generate debug symbols. If you want to debug your program, this setting must be activated. Xcode's Debug build configuration initially sets the Generate Debug Symbols build setting to Yes so you shouldn't have to change this build setting to debug your project.

Language

The Language collection contains miscellaneous compiler settings that don't fit anywhere else. Some of the more interesting things you can do in the Language collection include the following:

- Choose the language compiler.
- Choose the language standard.
- Enable exception handling.
- Set compiler flags.
- Enable Objective-C automatic reference counting.

Choosing the Language Compiler

The Compile Sources As setting is where you tell Xcode the language compiler to use: C, C++, Objective-C, or Objective-C++. Initially Xcode compiles your program's files according to the extension you give the files. Files with the `.c` extension use the C compiler. Files with the `.cpp` extension use the C++ compiler. Files with the `.m` extension use the Objective-C compiler, and files with the `.mm` extension use the Objective-C++ compiler. If you're writing code in C you may want to compile your programs with the C++ compiler. The C++ compiler can compile C code, and it's stricter than the C compiler. Compiling your C code with the C++ compiler can catch errors you wouldn't find with the C compiler.

Choosing the Language Standard

The C Language Dialect setting is where you tell Xcode the language standard you want to use. The ANSI group defines the standards for the C and C++ languages. GNU (the group in charge of the GCC compiler) adds extensions to these standards. Periodically (normally every 5-10 years) the ANSI group updates the language standards. Use the C Language Dialect setting to specify the language standard you want Xcode to use to compile your program. C++ programs should use the C++ Language Dialect setting.

Enabling Exception Handling

The Enable C++ Exceptions setting turns on exception handling for C++ programs, and the Enable Objective-C Exceptions setting turns on exception handling for Objective-C programs. Exception handling is a way for C++ and Objective-C programs to handle errors that occur when the program is running. If your program uses `try`, `catch`, and `throw` statements, make sure you turn on exception handling.

Setting Compiler Flags

When you configure a compiler setting from the build settings editor, you're indirectly setting command-line compiler flags. By using the build settings editor to configure compiler settings, Xcode saves you from having to enter dozens of compiler flags. Xcode provides a lot of compiler settings for you, but if you need to set a compiler flag that Xcode doesn't supply a setting for, use the Other C Flags and Other C++ Flags settings. C and Objective-C programs should use the Other C Flags setting, and C++ and Objective-C++ programs should use the Other C++ Flags setting.

To set a compiler flag, select the appropriate setting and double-click the target or project column. A pop-up editor opens. Click the + button to add a flag.

Enabling Objective-C Automatic Reference Counting

Set the Objective-C Automatic Reference Counting build setting to Yes to enable automatic reference counting. Automatic reference counting provides automatic memory management of Objective-C objects, which simplifies memory management. You must use the LLVM compiler to use automatic reference counting. Automatic reference counting is supported on Mac OS X 10.6 and later and iOS 4 and later. If you use automatic reference counting in a Mac application, Xcode can build only a 64-bit version of the application. You will get a build error if you try to build a 32-bit version of the application.

Preprocessing

The Preprocessing collection is where you define and set the values for preprocessor macros. A macro performs text substitution. The following macro:

```
#define ARRAY_SIZE 500
```

Defines a macro `ARRAY_SIZE` that is a substitute for the value 500. Before compiling your program, the preprocessor goes through your code and replaces every instance of `ARRAY_SIZE` with the value 500.

The Preprocessing collection is for conditional macros. Suppose you add code to your program that writes debugging information to a file. As you're developing the program, you want to write the debugging information. When you have the code working properly, you want to stop writing the debugging information so your program will run faster. By defining a macro you can easily turn the debugging information on and off.

```
#define DEBUG

#ifdef DEBUG
    write debugging stuff
#endif
```

Instead of defining `DEBUG` in your code, you would define it in the Preprocessing collection for the Debug build configuration. Defining `DEBUG` tells Xcode to write the debugging information. Removing the `DEBUG` definition tells Xcode to skip over the debugging code. By defining `DEBUG` in the Preprocessing collection, you can set your project so the debug build writes the debugging information and the release build doesn't.

Warnings

The Warnings collection provides a series of build settings that tell the compiler when to generate warnings. Compiler warnings tell you when you're doing something that's syntactically correct, but probably wrong. Turning on warnings is a good idea during development because the warnings let the compiler tell you about possible problems in your code that would be difficult to find without the warnings. Compiler warnings can save you hours during debugging.

Setting the Treat Warnings as Errors build setting to Yes tells Xcode to treat compiler warnings as compiler errors. If your code generates any compiler warnings, Xcode won't compile the program, forcing you to fix the warnings to build the final product. Treating warnings as errors forces you to write clean code from the start.

If you want to turn off warnings, set the Inhibit All Warnings build setting to Yes. I don't recommend turning off all warnings because it allows problems in your code to slip past the compiler.

If there are warning flags that Xcode doesn't provide a setting for, use the Other Warning Flags build setting to add them.

Data Model Version Compiler

Only Core Data projects have a Data Model Version Compiler collection. The build settings in this collection suppress warnings from the compiler when it compiles your Core Data data models. You probably won't need to modify these build settings.

Interface Builder Compiler

You must have a xib file in your project to have the Interface Builder Compiler collection. A xib file is a nib file that is saved in XML format so version control systems can track the changes. When you build your project, the Interface Builder compiler, `ibtool`, compiles the xib file into a nib file and copies the nib file into your application bundle.

Xcode's initial settings work well for basic compiling; many of you won't ever need to change the initial settings. But if you need to customize the compilation, you must add compiler flags. If you add a flag that Xcode doesn't supply a build setting for, enter the flag in the Other Interface Builder Compiler Flags build setting.

Unless you're using third-party Interface Builder plug-ins, you shouldn't have to modify the other Interface Builder compiler build settings. If you are using plug-ins, enter them in the Plug-Ins build setting. Enter any plug-in search paths in the Plug-In Search Paths build setting.

iOS projects with a storyboard have an Interface Builder Storyboard Compiler section, which contains similar build settings for storyboards.

Apple removed the Interface Builder Compiler collection in Xcode 4.4.

Static Analyzer

The Static Analyzer collection contains build settings that control what the Clang static analyzer checks. Read the "Static Analysis" section later in this chapter for more information on the Clang static analyzer. iOS projects may not have the Static Analyzer collection.

Conditional Build Settings

Conditional build settings are settings that apply to a particular architecture or SDK. Suppose you're writing a Mac application and your code has some `#ifdef` statements that run different code for the 32 and 64-bit versions of the application.

```
#ifdef 64_BIT
    // 64-bit version of the code
#else
    // 32-bit version
#endif
```

You would add a conditional build setting to the Preprocessor Macros build setting for the 64-bit Intel version that defines the `64_BIT` macro.

To add a conditional build setting, select the build setting where you're adding the condition. Click the disclosure triangle next to the build setting to show the build configurations. Select a build configuration. Click the + button next to the build configuration name to add a conditional build setting.

When you add a conditional build setting, a pop-up cell appears underneath the build setting. Use the pop-up cell to specify the condition. There are two types of pop-up cells for conditional build settings. The first type has the initial value Any SDK and lets you choose the SDK the build setting value will apply to. The second type has the initial value Any Architecture | Any SDK and lets you choose the architecture and SDK. Not all build settings let you choose the architecture. Set the value of the build setting after choosing the SDK and architecture.

To delete a conditional build setting, select it and click the minus button next to the setting.

Adding Your Own Build Settings

After looking at all the build settings Xcode has, you might be wondering why you would need to add your own build settings. The build settings editor's list of build settings, while impressive, is not an exhaustive list. If you need to change a build setting that is not in the list, you must add the build setting.

You're more likely to create conditional build settings than define new build settings. The main reason to add a build setting occurs if your project has a Run Script build phase. You may want to define a build setting to use in your script.

To add a build setting, click the Add Build Setting button in the lower right corner of the build settings editor. A menu will open. Choose Add User-Defined Setting. The new setting resides in the User-Defined collection. It initially has the name New_Setting. Select the setting and either press the Return key or click the mouse button a second time to rename the build setting. The name of a build setting can contain letters, underscore characters, and numbers; no spaces allowed. You cannot start a build setting name with a number.

Double-click the column next to the build setting to give it a value. A pop-up editor opens. Enter the value and press the Return key. Click the disclosure triangle next to a build setting to enter different values for the Debug and Release build configurations.

To remove a build setting you created, select it and press the Delete key.

Configuration Settings Files

A *configuration settings file* is a text file that contains build settings. If you find yourself changing the same build settings for all your projects, you should use a configuration settings file. Add the settings you're always changing to the configuration settings file. Add the configuration settings file to your project and tell Xcode to base your build configurations on the configuration settings file. Xcode will use the settings in the configuration settings file to build your project so you don't have to change the settings in the build settings editor.

Creating a Configuration Settings File

Because a configuration settings file is just a text file, you can create one in any text editor. Give the file the extension `.xcconfig` so Xcode knows the file is a configuration settings file. The easiest way to create a configuration settings file is to create it in Xcode. Choose File > New > File to create a new file. Select Configuration Settings File from the list of file types. You can find the configuration settings file in the Other group under Mac OS X and iOS.

When you create a new file, Xcode sets things up so the file is added to your project's targets. Do not add configuration settings files to targets. Deselect the checkbox next to each target in your project before clicking the Create button to create the file.

Creating a configuration settings file in Xcode is easy, but you're not going to want to create a new configuration settings file for each project you create. The point of using a configuration settings file is to create a list of build settings once and use that list in multiple projects. Choose File > Add Files to ProjectName to add your configuration settings file to other projects. Make sure you do not add the configuration settings file to the other projects' targets.

Xcode projects can contain multiple configuration settings files. You can have one configuration settings file of debug build settings, a second configuration settings file of release build settings, and use both files in your project. Use the debug configuration settings file in your Debug build configuration, and use the release configuration settings file in your Release build configuration.

What Goes in a Configuration Settings File?

A configuration settings file contains a list of build settings, with one setting per line. Each setting takes the following form:

```
SETTING = value;
```

Selecting a build setting in Xcode's build settings editor shows the formal build setting name as the declaration in Quick Help. Choose View > Utilities > Show Quick Help Inspector to open Quick Help. Build settings usually contain all uppercase letters. If you select the Optimization Level build setting, which is part of the Code Generation build settings collection, you will see the setting's formal name is `GCC_OPTIMIZATION_LEVEL`. If you don't want to open Quick Help, you can see the formal build setting names in the project editor by choosing Editor > Show Setting Names. You can also read the *Xcode Build Setting Reference* to see a list of available build settings. The *Xcode Build Setting Reference* is part of the Xcode documentation.

Suppose you want your projects to use Clang LLVM 4.0 as the compiler. You would add the following setting to the configuration settings file:

```
GCC_VERSION = com.apple.compilers.llvm.clang.4_0;
```

If you want to add comments to a configuration settings file, remember that Xcode uses the C++ `//` comment style for configuration settings files.

```
// This is a comment.
```

A configuration settings file can contain as many build settings as you want, but don't put every build setting in the configuration settings file. Put only the build settings you don't want to be constantly changing. If there are only three build settings you're constantly changing, put those three settings in the configuration settings file. Xcode will use the settings in the build settings editor for the rest of the build settings.

Telling Your Project to Use a Configuration Settings File

After writing your configuration settings files, you must add them to your project and tell Xcode what configuration settings file to use. If you haven't added configuration settings files to your project, choose File > Add Files to ProjectName to add them, making sure you don't add them to your project's targets.

Select the project file from the project navigator to open the project editor. Select the project from the project editor. Click the Info button at the top of the editor. In the list of build configurations is a column called Based on Configuration File. Use that column to set a configuration settings file for that build configuration. For a given build configuration, you have the option to use a configuration settings file for the project or for a specific target.

Overriding the Configuration Settings File

Suppose you have a configuration settings file with 15 settings. You create a new project, but want to use only 13 of the 15 settings in the configuration settings file. How do you tell Xcode to use only the 13 settings you want to use?

The solution is to use the build settings editor. When you change a build setting from the build settings editor, it overrides the setting in the configuration settings file. In the example from the last paragraph, you would open your project's build settings editor and change the two settings you want to override.

Because the build settings editor overrides the configuration settings file, be careful when you open the build settings editor. You don't want to accidentally override any of your configuration settings file's build settings.

Compiling Your Program

After writing the source code for your project, you must compile the code to create the final product and make sure the code works. Xcode calls the process building. When building your project, Xcode compiles your source code files, links them with the frameworks in your project, and creates the final product, such as an application or a library.

Schemes

Schemes allow you to configure the way Xcode builds, runs, tests, profiles, analyzes, and archives the targets in your project.

Choosing a Scheme

Use the Scheme menu in the project window toolbar to choose a scheme and a run destination. The Scheme menu is a path control that lets you pick the scheme and run destination separately. Xcode initially provides a scheme for each target in the workspace, which means you don't have much to choose from if you have a project with only one target.

The run destination controls how Xcode runs your project when you click the Run button. For an iOS application you can run on a device, the iPhone Simulator, or the iPad Simulator. Mac projects have two possible run destinations: 32-bit and 64-bit, but most projects initially have only the 64-bit run destination. You must set the Architectures build setting to build for both 32-bit and 64-bit Intel to have a choice of run destinations.

Workspaces have their own separate schemes. A project in a workspace has at least two schemes: one in the workspace and one for the project. Open the project outside of the workspace to access the project's scheme.

Opening the Scheme Editor

Choose Edit Scheme from the Scheme menu to open the scheme editor, which you can see in Figure 6.1. At the top of the scheme editor are controls that let you choose the scheme, choose the run destination, and turn on breakpoints. The Breakpoints button toggles running and debugging your project. On the left side of the scheme editor is a list of scheme actions: Build, Run, Test, Profile, Analyze, and Archive.

One thing you'll notice for every scheme action besides Build is that you can set the build configuration to use when performing that action. This keeps you from having to constantly set the build configuration. The Run action is the only action where you would need to change the build configuration often.

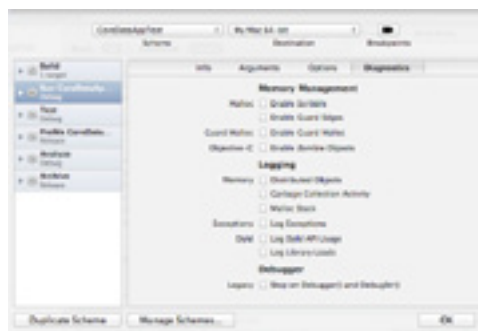


Figure 6.1

Scheme editor

Build

The Build action of the scheme editor has two sections: Build Options and Targets. The Build Options section has two checkboxes: Parallelize Build and Find Implicit Dependencies. Selecting the Parallelize Build checkbox tells Xcode to build independent targets in parallel, which can make building go faster on a multi-core Mac. But most Xcode projects won't be able to take advantage of the Parallelize Build checkbox because most Xcode projects don't have multiple independent targets. Selecting the Find Implicit Dependencies checkbox tells Xcode to find and build target dependencies automatically.

The Targets section contains a list of targets to build. Click the + button to add a target. For each target there is a series of checkboxes to analyze, test, run, profile, and archive the project. Select the checkbox for an action to enable that action for the target. If you want to profile the target in Instruments, make sure the Profile checkbox is selected.

Run

The Run action lets you control how Xcode runs your project. It has the following groups, which you can access using the buttons in the upper part of the scheme editor:

- Info
- Arguments
- Options
- Diagnostics

Use the Info group to choose the build configuration, choose the executable to launch, choose the debugger to use, and determine whether or not to launch the debugger automatically. You would change the executable if you were writing a framework and wanted to run a test application that tested the framework.

The Arguments group lets you add command-line arguments and set environment variables for your project. Before you set environment variables, look at the Diagnostics group, which has checkboxes for some commonly used environment variables.

Some Xcode projects have a Module Names To Load Debug Symbols For section in the Arguments group. Use this group to tell Xcode to load debug symbols for frameworks and libraries that link to your application when you debug the application. Read the section “Configuring Your Scheme for Debugging” in Chapter 7, “Debugging”, for additional information.

The Options group contains dynamic options that depend on the platform and operating system. One of the options for a Mac application is to set a custom working directory for your project. iOS applications can use the Options group to allow location simulation and to supply an application data package for the application to use. Not every project has an Options group.

The Diagnostics group contains checkboxes that can help you debug your project. Read the section “Configuring Your Scheme for Debugging” in Chapter 7, “Debugging”, for detailed information on the checkboxes in the Diagnostics group.

Test

The Test action is for unit testing. Your project must have a unit testing target for the Test action settings to have any effect. The most important thing you can do in the Test action is control the tests that are run when you test your project. Xcode is initially set to run all tests. Use the disclosure triangles and checkboxes to disable groups of tests.

Other things you can do in the Test action include choosing the debugger to use to debug unit tests, passing separate command-line arguments for the test, passing separate environment variables for the test, and supplying an application data package for testing. Initially the unit test uses the same arguments and environment variables the application target uses.

Profile

The Profile action lets you profile your application in Instruments. Choose Product > Profile to profile the application. You can choose the build configuration, the executable, the instrument to use, and a custom working directory for the executable. There’s also an Arguments group, which lets you pass command-line arguments and set environment variables to use when running in Instruments.

If you choose Ask on Launch from the Instrument pop-up menu, Instruments asks you what instrument to use when you choose Product > Profile. Otherwise, Instruments launches and traces your program using the instrument you chose from the Instrument pop-up menu. Read Chapter 9 to learn more about Instruments.

Analyze

There is not much to configure for the Analyze action. You specify the build configuration to use when running your code through the static analyzer.

Archive

The Archive action archives your application so you can distribute your application to testers, verify the application before submitting to the App Store, and submit the application to the App Store.

Xcode should be set to use the Release build configuration for archiving, but you can choose a build configuration from the Build Configuration menu if you need to. Enter the name of the archived application in the Archive Name text field. Selecting the Reveal Archive in Organizer checkbox tells Xcode to show the archive in the Organizer after archiving your project.

Pre and Post-Actions

Each scheme action has a disclosure triangle next to it in the scheme editor. Clicking the disclosure triangle lets you specify pre-actions and post-actions for a scheme action. Setting up test data is a situation where you would use a pre-action. Uploading an archive to a server is a situation where you would use a post-action.

To add a pre-action or post-action to a scheme action, click the disclosure triangle next to the scheme action. Select Pre-actions or Post-actions. The list of pre-actions and post-actions is initially empty. Click the + button to add a pre-action or post-action. You can run a shell script or send an email in a pre-action or post-action. If a shell script is related to the build product itself, you should use a Run Script build phase instead of pre-actions or post-actions.

Adding and Managing Schemes

When you create a project Xcode creates a scheme for you. While the included scheme is enough in many situations, sometimes you need to add a scheme. Choose New Scheme from the Scheme menu in the project window toolbar to add a scheme to your project. A sheet opens. Choose a target, name the scheme, and click the OK button to create the new scheme.

Choosing Manage Schemes from the Scheme menu opens a sheet to manage your project's schemes. Most of the sheet contains a scheme list. The scheme list has four columns: Show, Scheme, Container, and Shared. Selecting the Show checkbox makes the scheme appear in the Scheme menu. Deselecting the checkbox can help if you're working on a team and don't want to see the other team members' schemes in your Scheme menu. The Scheme column lets you rename the scheme. The Container column can be either a project or a workspace. The Shared checkbox lets you share a scheme with other team members.

Below the scheme list are buttons to add a scheme, delete a scheme, duplicate a scheme, import a scheme, and export a scheme.

Above the scheme list is an Autocreate schemes checkbox. Selecting this checkbox tells Xcode to create a new scheme when you add a target to your project. Autocreating schemes is generally a good thing, but you may not want to create a new scheme when you add a unit testing target because you can unit test from the application target's scheme. Clicking the Autocreate Schemes Now button tells Xcode to autocreate the schemes if you don't have the Autocreate schemes checkbox selected.

Precompiled Headers

Xcode supplies a prefix file for Cocoa and iOS application projects. The prefix file contains a list of header files. Xcode precompiles the header files in the prefix file. By precompiling these header files, Xcode has less work to do when it compiles the files that include the precompiled header files. If you have a lot of source code files that include a header file, adding the header file to the prefix file speeds up the building of your project.

What header files should you put in a prefix file? Header files that multiple source files include and header files that don't change often are the best files to add to a prefix file. The header files from Apple's frameworks are great files to put in a prefix file. If you're writing a Cocoa program, most of your files are going to include the Cocoa header file `Cocoa.h`, and you're not going to make changes to `Cocoa.h`.

Why should you place header files that rarely change in a prefix file? Xcode recompiles the precompiled header file when the prefix file changes or the header files in the prefix file change. If you add a header file that you're constantly changing, Xcode has to recompile the precompiled header every time you change the header file. Recompiling the precompiled header is slower than not using prefix files at all.

To add header files to a prefix file, open the prefix file. The prefix file of a Cocoa application project has the name `ProjectName-Prefix.pch` and includes the Cocoa header file. Use the `#import` (Objective-C) or `#include` (C and C++) statement to add other header files.

After adding files to your prefix file, make sure Xcode is set to precompile the headers in the prefix file.

1. Select the project file from the project navigator.
2. Select the target from the project editor.
3. Click the Build Settings button at the top of the editor.
4. Make sure the Precompile Prefix Header build setting is set to Yes.
5. Make sure the Prefix Header build setting shows the name of your prefix file.

The Precompile Prefix Header and Prefix Header build settings are part of the Language collection.

Cleaning Targets

Cleaning a target removes everything you previously built for a target. When you build a project with a clean target, you must recompile all your source code files so you shouldn't clean a target unless it's necessary. Changing compiler settings in your project requires a clean target for the changes to take effect. Suppose you're compiling your files with the Fast optimization level and you change the optimization level to Faster to measure the speed difference between the two levels. To increase the optimization level you must recompile all the files, which requires you to clean the target.

Choose Product > Clean to clean the active target. If you hold down the Option key, the Clean menu item in the Product menu changes to Clean Build Folder, which removes everything in the build folder.

Building Your Project

Xcode provides the following ways to build your projects:

- Build a project.
- Build a project and run it in Xcode.
- Build a project and run unit tests.
- Build a project and profile it in Instruments.
- Build and analyze.
- Build and archive.

Choose Product > Build to build your project without running it. Xcode compiles any source code files that changed since the last time you built the project. If you haven't built your project before or you cleaned the target, Xcode compiles all the source code files. After compiling the files, Xcode performs any additional steps required to create the target.

Choose Product > Run to build your project and run it. Building and running builds your project and runs it in Xcode. To be able to run your program, it must build successfully with no errors. The Breakpoints button in the project window toolbar toggles running and debugging your program. Choosing Product > Perform Action > Run Without Building runs your program without building it.

Choose Product > Test to build your project and run unit tests. You must have a unit testing target in your project to be able to run unit tests from Xcode. If you set the Test After Build build setting to Yes, choosing Product > Build For > Testing will also build your project and run unit tests. Choosing Product > Perform Action > Test Without Building runs the unit tests without building the project.

Choose Product > Profile to build your project and profile it in Instruments. Xcode initially is set to ask you what instrument to use when your profile, but you can edit your scheme to always use a particular instrument when profiling your project. Choosing Product > Perform Action > Profile Without Building profiles your project without building the project.

Choose Product > Analyze to build your project and run the code through the Clang static analyzer. Read the “Static Analysis” section later in this chapter to learn more about the static analyzer.

Choose Product > Archive to build your project and create an archive for submission to the App Store.

You can also use the Run button to run, test, profile, and analyze your project. Click the Run button, hold the mouse button down, and choose an action from the menu that opens.

Where’s My Application?

You may be wondering where to find your application after building it. The simplest way to find your application is to click the disclosure triangle next to the Products folder in the project navigator. Clicking the disclosure triangle shows the project’s build products, one of which is your application. Select the application, right-click, and choose Show in Finder.

Xcode initially stores the derived data for projects, including build products, in the following location:

`/Users/Username/Library/Developer/Xcode/DerivedData`

If you want your build products to go in a different location, open Xcode’s Locations preferences. There are two alternatives to Xcode’s default location: relative and custom. If you choose a relative location, Xcode places the derived data in a subfolder of your project or workspace folder. If you pick a custom location, there is a button on the right side of the text field. Clicking it opens an Open panel that lets you pick where the derived data goes. You can choose the derived data location for an individual project by choosing File > Project Settings.

Seeing More Build Details

When you build your project, the only thing Xcode tells you is whether the build succeeded or failed. If the build failed, you want to know what caused the build to fail. Open the issue navigator to see a list of all errors, warnings, and analyzer issues. Selecting an issue from the issue navigator opens the file in the editor and takes you to the line of code where the issue occurred.

There is a set of controls at the top of the issue navigator that determines how Xcode groups the issues in the navigator: by file or by type. When ordering by file, the issue navigator groups the issues by the source file where the issues occurred. When ordering by type, the issue navigator groups the issues by issue. Suppose you have multiple warnings in your project for having unused variables in your code. Grouping the issues by type places all the unused variable warnings in a group.

Message Bubbles

When there is an error, warning, or analyzer issue in your code, a message bubble appears in the line of code in the editor. The line of code is highlighted. The gutter has an icon that identifies the issue: error, warning, or analyzer issue. Clicking the icon hides the message bubble; click the icon again to show the message bubble. On the right side is the error message. If there are multiple errors in a line of code, a number appears to the right of the error message. Click the number to see all the errors for that line of code.

Opening the Build Results Window

To see more details of a build, open the build results window, shown in Figure 6.2, by opening the log navigator and selecting a build from the log list. The most recent build appears at the top of the list.

The build results area gives you a play-by-play account of the build, listing each step taken during the build. If the step was successful, it has a green checkmark next to it. Warnings have a yellow icon next to them, and errors have a red stop sign icon next to them. Errors and warnings will also have a message detailing what went wrong. Static analyzer issues have a blue icon next to them with a message detailing a problem. For more information on static analyzer issues, refer to the “Static Analysis” section later in this chapter. Double-clicking a step opens the file in a separate window.

Showing the Build Transcript

Each build step executes commands from the command line. Xcode shields you from the details of executing commands, but the build transcript shows the executed commands. When you compile a file, the build results area tells you that Xcode compiled the file. The build transcript shows the commands Xcode called to compile the file.

Selecting a build step from the build results window makes the build transcript button appear on the right side of the window. Clicking the build transcript button shows the build transcript for that step. If the step has an error or warning, the build transcript button will appear without you having to select the step.

Filtering the Build Results

At the top of the build results window is a scope bar with two sets of buttons that lets you filter what appears in the build results window. The first set determines whether to show all results or the latest results. All results shows the results of each file in the project while the latest results shows only the files that were compiled during the most recent build.

The second set of buttons determines what messages get displayed in the build results window. You can display all messages, all issues, or errors only. Displaying all messages shows every step in the build process. Displaying all issues displays any build problems: errors, warnings, and analyzer issues. Displaying errors only displays only compiler errors, linker errors, and unit test failures.

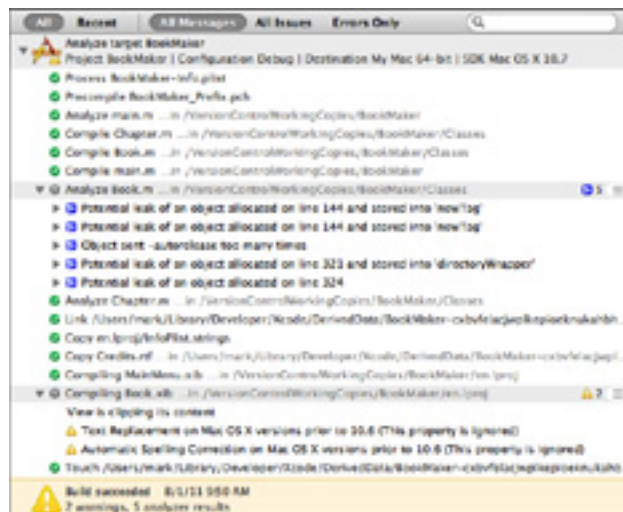


Figure 6.2

Build results window

Customizing Xcode Behaviors

Xcode has a set of behaviors for conditions that occur in your project, such as a build failing. When a build fails, Xcode initially shows an alert that tells you the build failed. You may want Xcode to behave differently when a build fails, such as opening the issue navigator so you can see the build errors. You can get the behavior you want by customizing Xcode's behaviors.

To customize behaviors open Xcode's Behaviors preferences by choosing Xcode > Behaviors > Edit Behaviors. On the left side of the preferences window is a list of behaviors, which mostly involve building, testing, and running your project. Select a behavior. You will see a series of checkboxes in the preferences window that represent the actions you can perform in a behavior. Some actions you can add to a behavior include playing a sound, showing a tab, and showing a specific navigator. Select a checkbox next to an action to add that action to the behavior.

Xcode also allows you to create custom behaviors, which you activate with a keyboard shortcut. Click the + button in the lower left corner of the preferences window to add a custom behavior. Name the behavior and press the Return key. Next to the behavior name is the Command key symbol. Double-click the symbol to enter the keyboard shortcut that activates the custom behavior. If your shortcut conflicts with an existing shortcut, an alert opens. Click the Replace button to overwrite the existing shortcut. Click the Cancel button to enter another shortcut for the custom behavior.

After setting the keyboard shortcut, use the checkboxes to add the behavior you want. When you invoke the keyboard shortcut, Xcode performs your custom behavior.

Tips for Correcting Build Errors

One of the most frustrating aspects of writing Mac and iOS software for people new to the Mac platform is getting the program to compile. Compilers can be picky, spitting out error messages on code that works on another compiler. In this section I provide some tips to fixing the errors you get when trying to build your project.

Add All Necessary Frameworks

Missing frameworks are the leading cause of linker errors and cause compiler errors if your program calls functions from the missing frameworks. Make sure you add the frameworks to your project.

If you're using libraries and frameworks in your project that Apple did not supply, Xcode may be unable to find them, even though you did add the libraries and frameworks to your project. You will have to add search paths for your libraries and frameworks. Refer to the "Search Paths" section earlier in this chapter for more information on adding search paths to your project.

Include Necessary Header Files

Adding a framework isn't the only step you must take if you want to call the framework's functions in your program. You must remember to include the framework's header files as well. Mac OS X and iOS have a slightly different way of including system headers than other operating systems. The method Mac OS X and iOS use to include system headers causes build errors if you're unaware of the method. Suppose you want to play QuickTime movies in your Cocoa application. You add the Quartz framework to your project and include the header file `Quartz.h`.

```
#include <Quartz.h>
```

This statement generates an error during building. Mac OS X and iOS require you to enter the framework the header file belongs to before the header file when including a system header. The following line demonstrates the proper way to include the `Quartz.h` header file:

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

The Error May Not Be Where Xcode Says It Is

When Xcode finds a syntax error in your code, it reports the file where the error occurs along with the line number. You go to the line of code and can't find anything wrong. In this situation the error may have occurred in an earlier line of code. Check the lines of code above the line where Xcode reported the error.

One Error Can Cause Multiple Syntax Errors

Sometimes one error can trigger multiple syntax errors. It can be overwhelming to look at the build transcript and see hundreds of error messages. Fix one error at a time, rebuilding the project after each fix. You may discover you have fewer mistakes in your program than you thought.

Look for Typographical Errors

Typographical errors cause a high percentage of syntax errors. Make sure you're not missing semicolons. Make sure each left brace has a matching right brace. Make sure each left parenthesis has a matching right one. Check for misspelled variable and function names.

Check Function Arguments

Improper function arguments cause many syntax errors. Check the arguments in your function calls. Make sure the number of arguments match, the arguments are in the right order, and the arguments have the proper data type. Pointer variables as arguments cause a lot of syntax errors. If you pass a non-pointer variable improperly, you get a syntax error.

Let's use an example to demonstrate passing a non-pointer variable improperly. The Core Foundation function `CFURLGetFSRef()` creates a file system reference from a file location. You would use the file system reference to open the file. The second argument to `CFURLGetFSRef()` is a pointer to a file system reference. If you have the following code:

```
CFURLRef theFile;  
FSRef fileRef;  
Boolean success;  
  
// Code to retrieve the file location has been omitted.  
  
success = CFURLGetFSRef(theFile, fileRef);
```

You get an error because the function `CFURLGetFSRef()` takes a pointer to `FSRef` as an argument and you passed a `FSRef`. You must pass the address of the variable `fileRef`.

```
success = CFURLGetFSRef(theFile, &fileRef);
```

Building for Unsupported Languages

Xcode natively supports projects written in AppleScript, C, C++, and Objective-C. It can also build projects written in languages Xcode doesn't natively support. Mac OS X includes Java, Perl, PHP, Python, and Ruby. Other popular languages are Haskell, Fortran, Lua, Pascal, and Smalltalk, but you have to download a compiler to use those languages. You can even create your own programming language and build programs written in that language from Xcode. All you need is a program to do the building.

To use a language that Xcode doesn't natively support, create an external build system project. You must tell Xcode to use your language's build tool to build the project.

1. Select the project name from the project navigator to open the project editor.
2. Select the target from the targets list on the left side of the project editor.
3. Click the Info button at the top of the project editor to set the build tool, which you can see in Figure 6.3.
4. In the Build Tool text field, type the path to the program you're going to use to build the program. Initially for an external build system project, the build tool is `make`, located at `/usr/bin/make`.
5. In the Arguments text field, enter any arguments you want to use to build the program. Normally this involves compiler flags you want to use when building the program.
6. If your program needs to set a working directory, enter it in the Directory text field or click the button on the right side of the text field to choose a working directory.
7. Selecting the Pass build settings in environment checkbox tells Xcode to pass the project's build settings to the build tool's environment.

When you build your project, Xcode uses the tool you specified to do the building.

Static Analysis

Xcode 4 comes with the Clang static analyzer. The static analyzer goes through your code and looks for bugs. It can find mistakes such as memory leaks, bad pointer references, dead code, and uninitialized variables. These are problems that can be difficult to find on your own.

To run the static analyzer, choose **Product > Analyze**. Activating the **Run Static Analyzer** build setting tells Xcode to run the analyzer every time it builds your project. Static analysis takes longer than compiling so you may not want to automatically run the analyzer on large projects.

The issue navigator lists the issues the analyzer found. The issue navigator should be visible on the left side of the project window. If you can't see the issue navigator, choose **View > Navigators > Show Issue Navigator**. The static analyzer provides more detailed information than a compiler error message. A compiler error message tells you the error and the line of code where the error occurred. The static analyzer shows all the steps in your code that led to the issue. Click the disclosure triangle next to an issue to see the steps.

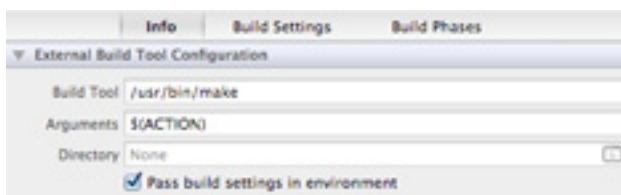


Figure 6.3

Setting the build tool for a non-native target

When you select an analyzer issue from the issue navigator, Xcode opens the file containing the issue in the editor and takes you to the line of code with the issue. From there you can locate the cause of the issue and correct it. Double-clicking an issue opens the file in a separate window.

Each analyzer issue has a message bubble in the editor. Xcode highlights the line of code where the issue occurred. In the gutter is an analyzer issue icon. Clicking the icon hides the message bubble; click the icon a second time to show the message bubble. To the right of the code is the analyzer message. If a line of code has additional steps, its icon contains two arrows: one facing up and one facing right. Click the analyzer icon to the left of the message to see the additional steps, which you can see in Figure 6.4. You'll notice the icon changes to a right-facing arrow.

When a line of code has additional steps, a scope bar appears under the editor's jump bar. Use the menu on the left side to pick a step to analyze, or use the left and right arrow buttons to navigate the various steps in the analyzer issue.

Generating Output Files

Xcode can create two types of output files from your source code files: assembly language and preprocessed. An assembly language output file contains the results of the compiler translating your source code into assembly language. A preprocessed output file contains the results of running the source code through the compiler's preprocessor.

How do you generate output files? Select a C, C++, or Objective-C file from the project navigator and choose Product > Generate Output. Use the appropriate submenu to choose the type of output file to generate.



Figure 6.4

Static analysis multi-step issue

When you generate an output file, Xcode displays it in the editor. Below the editor is a pop-up menu that lets you specify the scheme action for Xcode to generate the output file. Initially Xcode generates the output file for running, but you can also choose testing, profiling, analyzing, or archiving.

Creating Applications that Run on iPhones and iPads

There are two ways to create an application that runs on both iPhones and iPads: create a universal binary or create two device-specific applications. An iOS universal binary runs on both iPhones and iPads. There are two ways to create a universal binary. Create a universal application project or take an iPhone project and upgrade it to a universal binary.

Creating a New Universal Project

The easiest way to create a universal iOS application is to choose Universal from the Devices menu when creating the project. If you're running an earlier version of Xcode and don't have a Universal option, choose iPhone and upgrade the project to add iPad support.

Upgrading an Existing iPhone Project

To upgrade an existing iPhone project to a universal one, select the project from the project navigator to open the project editor. Select the target in the project editor. Click the Summary button at the top of the editor. Choose Universal from the Devices pop-up menu.

Universal Application Build Settings

When upgrading the iPhone application to a universal application, Xcode should have modified the build settings to support a universal application, but you should still check the Targeted Device Family build setting and make sure it is set to iPhone/iPad. Remember to check the build setting for the target, not the project. Target build settings override project build settings.

If you are supporting older devices that have the `armv6` architecture, you will have to add `armv6` to the Architectures build setting. Choose Other and add `armv6` to the list of architectures.

Creating Two Device-Specific Applications

Instead of having a universal application, you may prefer to have two versions of your application: one for iPhone and one for iPad. To create two device-specific applications, start by creating an iPhone application project. Select the target from the project editor, right-click, and choose Duplicate. An alert opens. Click the Duplicate and Transition to iPad button. Xcode adds an iPad application target to the project as well as a Resources folder for the iPad version that contains iPad versions of the project's xib files.

Chapter 7

Debugging

Writing software is difficult. Unless you're writing a very simple program, your code will have errors the compiler and static analyzer won't catch. Finding the errors by running your program can be difficult because the program runs too fast. It's like watching a DVD in fast forward mode.

A debugger lets you slow things down so you can see what is happening when your program runs. By using a debugger you can pause your program at any line of code, step through each line of code, and examine the values of your program's variables. A debugger helps immensely when you're trying to find out what's wrong with your code. For those of you new to programming, learning to use a debugger is an important skill to learn. In this chapter you'll learn how to use Xcode's debugger.

Before You Debug

If you create a new project, write the code for the project, and build the project, you should be able to debug your program with no problems. But there are steps you can take before you debug to improve your debugging experience.

Configuring Your Scheme for Debugging

Most of the work in setting up your project for debugging takes place in the scheme editor. Choose Edit Scheme from the Scheme menu in the project window toolbar to open the scheme editor. Select Run from the list of actions on the left side of the scheme editor. The scheme editor has the following buttons at the top of the editor for the Run action: Info, Arguments, Options, and Diagnostics. Some projects do not have an Options button.

Info

The Info group has pop-up menus to choose the build configuration, executable, and debugger. Set the build configuration to Debug. Make sure your application is the choice in the Executable menu. Choose a debugger from the Debugger menu.

Xcode projects have two debugger choices: GDB and LLDB. LLDB is faster than GDB and is the future of Xcode debugging, but LLDB is newer than GDB. If you find something doesn't work right in Xcode's debugger, it may mean that feature has not yet been implemented in LLDB. Switch to GDB and see if things improve.

You have two debugging launch options: launch the debugger automatically or wait until you launch the application before launching the debugger. In most cases you should launch the debugger automatically. The most common reason to not launch the debugger automatically is if you have an issue when your application starts up.

Xcode 4.4 adds the option to debug your project as yourself or as root. In most cases you should debug as yourself. If you are debugging a root process that requires root access, select the root radio button.

Arguments

The Arguments group lets you set environment variables. Mac OS X ships with a set of environment variables that can help you debug your application. The section "Setting Environment Variables for Debugging" later in this chapter has detailed information on the environment variables that help during debugging.

Some Xcode projects have a Module Names To Load Debug Symbols For section in the Arguments group. Use this section to tell Xcode to load debug symbols for frameworks and libraries that link to your application when you debug the application. The frameworks and libraries must have debug symbols to be able to load them. Click the + button to add a module listing. Each listing has two columns: Criteria and Module Name or Pattern. The criteria can be one of the following: contains, starts with, or ends with. Use the pop-up cell to choose a criteria. Enter a name in the Module Name or Pattern column.

Options

The Options group contains dynamic options that depend on your project's platform. If you're writing a document-based Cocoa application that supports Mac OS X 10.7's document versioning, selecting the Allow debugging when using document Versions Browser checkbox lets you debug when browsing versions of a document.

Diagnostics

The Diagnostics group contains options to help diagnose hard to find bugs. The scheme editor groups these options into three sets of checkboxes: Memory Management, Logging, and Debugger.

Memory Management

The Memory Management set of checkboxes has four checkboxes: Enable Scribble, Enable Guard Edges, Enable Guard Malloc, and Enable Zombie Objects. Selecting the Enable Scribble checkbox turns on the `MallocScribble` environment variable. Turning on the `MallocScribble` environment variable tells the operating system to fill freed memory with garbage values to keep your application from accessing the memory again.

Selecting the Enable Guard Edges checkbox turns on the `MallocGuardEdges` environment variable. Turning on the `MallocGuardEdges` environment variable tells the operating system to place guard buffers on each edge of a large memory buffer. If your application reads or writes past the memory buffer, the application crashes.

Selecting the Enable Guard Malloc checkbox tells Xcode to use the Guard Malloc library when debugging your project. Memory errors in your code are difficult to debug because the problems seem to occur randomly. Your program may run well one time and crash the next time you run it. The Guard Malloc library eliminates the randomness. When your program commits a memory access error, Guard Malloc crashes the program, which helps you locate the source of the error.

The downside of using Guard Malloc is that your program runs up to 100 times slower when running with Guard Malloc. Because of the slowdown, I recommend not using Guard Malloc the first time you debug your program. Correct the errors you find without Guard Malloc before debugging with Guard Malloc.

Selecting the Enable Zombie Objects checkbox tells Xcode to create zombie objects when an object's retain count reaches zero. Zombie objects alert you when you try to access an object you freed. Accessing freed objects is a common cause of crashes in Objective-C applications, and enabling zombie objects make finding the cause of these crashes easier.

Logging

The Logging set of checkboxes has checkboxes to log activity. By selecting the appropriate checkboxes, you can tell Xcode to log distributed objects, garbage collection activity, exceptions, Dyld (Dynamic Linker) API usage, and library loads when debugging your project.

Selecting the Malloc Stack checkbox turns on the `MallocStackLogging` environment variable. Turning on the `MallocStackLogging` environment variable tells the operating system to log the call stack when your application makes a memory allocation.

Debugger

The Debugger set of checkboxes consists of one checkbox. Selecting that checkbox tells the debugger to stop on calls to `Debugger()` and `DebugStr()`. `Debugger()` and `DebugStr()` are old Carbon functions that print log messages to the console. Cocoa and Cocoa Touch developers should use `NSLog` instead of `Debugger()` or `DebugStr()`. You can ignore this checkbox unless your code happens to call `Debugger()` or `DebugStr()`.

Setting Environment Variables for Debugging

Mac OS X's malloc library has a collection of environment variables to help debug memory allocations. Table 7.1 lists the variables. To set environment variables in Xcode:

1. Open the scheme editor by choosing Edit Scheme from the Scheme menu in the project window toolbar.
2. Select Run from the list on the left side of the scheme editor.
3. Click the Arguments button at the top of the scheme editor.
4. To add an environment variable, click the + button in the section Environment Variables.
5. Give the environment variable a name and a value. Giving an environment variable a value of zero disables the variable.

The Diagnostics group of the scheme editor has checkboxes to set the `MallocScribble`, `MallocGuardEdges`, and `MallocStackLogging` environment variables.

Table 7.1 Malloc Library Environment Variables

Variable Name	Value	Description
<code>MallocStackLogging</code>	Any integer > 0	Logs the chain of functions your program called to make the memory allocation.
<code>MallocStackLoggingNoCompact</code>	Any integer > 0	Does what <code>MallocStackLogging</code> does and remembers call stacks of memory allocations that no longer exist. Set this variable to remember every memory allocation made at a certain memory address.
<code>MallocScribble</code>	Any integer > 0	When your program frees memory, the operating system fills the memory with garbage values so your program can't accidentally access the memory again.
<code>MallocGuardEdges</code>	Any integer > 0	For large (over 4096 bytes) buffers the operating system places guard buffers on each edge of the buffer. If your program tries to read or write past the buffer, the program will crash.
<code>MallocDoNotProtectPrelude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer in front of the buffer.
<code>MallocDoNotProtectPostlude</code>	Any integer > 0	Works with <code>MallocGuardEdges</code> . The operating system won't place a guard buffer behind the buffer.
<code>MallocCheckHeapStart</code>	The number of allocations before checking the heap.	Checks the heap for corruption after <code>x</code> memory allocations, where <code>x</code> is the value you supply to <code>MallocCheckHeapStart</code> .

<code>MallocCheckHeapEach</code>	The interval to check the heap after the initial heap check.	Works with the <code>MallocCheckHeapStart</code> variable. After making the initial heap check with <code>MallocCheckHeapStart</code> , the operating system checks the heap every <code>x</code> memory allocations, where <code>x</code> is the value you supply to <code>MallocCheckHeapEach</code> .
<code>MallocCheckHeapSleep</code>	The number of seconds to sleep.	Your program goes to sleep when a heap corruption occurs.
<code>MallocCheckHeapAbort</code>	Any integer > 0	Aborts your program when a heap corruption occurs.
<code>MallocErrorAbort</code>	Any integer > 0	Aborts your program when it illegally frees memory or when an error occurs in a call to <code>malloc()</code> or <code>free()</code> .
<code>MallocCorruptionAbort</code>	Any integer > 0	Works similarly to <code>MallocErrorAbort</code> , but <code>MallocCorruptionAbort</code> does not abort when an out of memory condition occurs.
<code>MallocLogFile</code>	Path to file.	Error messages are written to a file instead of the console.

Choosing a Debugging Format

If you look at the Debug Information Format build setting (look in the Build Options collection), you will see the following debugging format options for C, C++, and Objective-C programs: DWARF, DWARF with dSYM File, and possibly Stabs. But Apple deprecated support for Stabs so you don't have much of a choice in debugging formats. You're going to be using DWARF. The decision to make is whether or not you want to create a dSYM file.

Should you decide to create a dSYM file, Xcode places all debugging symbols in a separate dSYM file. Using DWARF without a dSYM places the debugging symbols in the object files, which increases the size of the executable file. DWARF with dSYM is a good choice for release builds because you get a smaller executable file, but you have the debugging symbols available in case you release a product and users report problems. Plus, you don't have to worry about stripping debugging symbols out of the executable. For debug builds,

the decision between DWARF and DWARF with dSYM is a matter of personal preference. Xcode projects initially use DWARF for debug builds and DWARF with dSYM for release builds.

Breakpoints

To get any real debugging done, you need to set breakpoints. Technically, you can click the Pause button in the debug area then step through your code. But Mac OS X and iOS applications spend most of their time waiting for user input. When waiting for input your program isn't doing anything so there's nothing to debug.

Breakpoints tell the debugger to pause execution of your program at a specific place in your program. From there you can step through the code to find out what's wrong. By using breakpoints you can focus on a small section of code.

Setting Breakpoints

The easiest way to set a breakpoint is to open one of your source code files and click the gutter next to the line of code where you want your program to pause. The gutter will have a dark blue arrow next to the line of code. If the arrow is gray instead of blue, click the Breakpoints button in the project window toolbar. Clicking a breakpoint arrow in the gutter disables the breakpoint. A disabled breakpoint is transparent (light blue) instead of dark blue. Click the disabled breakpoint arrow to enable the breakpoint. Drag the arrow out of the gutter to delete a breakpoint.

If you want to see a list of all the breakpoints you've set, open the breakpoint navigator by choosing View > Navigators > Show Breakpoint Navigator. The breakpoint navigator separates the breakpoints by file and tells you the following information for each breakpoint:

- The type of breakpoint, represented by an icon: M for a file line breakpoint, Ex for an exception breakpoint, and the Greek letter sigma for a symbolic breakpoint.
- The function where the breakpoint resides.
- The line number in the file for file line breakpoints.
- Whether or not the breakpoint is enabled.

At the bottom of the breakpoint navigator is a + button. Click the + button to add symbolic and exception breakpoints. A *symbolic breakpoint* pauses your program when it enters the function where you added the breakpoint. An *exception breakpoint* pauses your program when an exception occurs. For C++ exceptions you can pause your code when it raises a specific exception. Otherwise your application pauses when it raises any exception.

To see more information on a breakpoint, select it, right-click, and choose Edit Breakpoint. A pop-up editor opens that shows six pieces of information for each breakpoint.

- The location of the breakpoint: file name and line number, symbol and module name, or exception name, depending on the type of breakpoint you set.
- A checkbox that tells you if the breakpoint is enabled.
- A condition that tells Xcode when to pause your program. A blank condition tells Xcode to pause your program every time you reach the breakpoint.
- An ignore count that tells Xcode to ignore the breakpoint the number of times that you specify.
- An action field that lets you add breakpoint actions.
- An automatically continue after evaluating checkbox that tells Xcode to continue execution after reaching the breakpoint.

The automatically continue checkbox requires some additional explanation. Deselecting the checkbox is the equivalent of pressing the Pause button on a DVD player. It pauses your program. If you want to step through your code after reaching the breakpoint, make sure the automatically continue checkbox is not selected.

Selecting the automatically continue checkbox is the equivalent of pressing the Pause button on a DVD player followed immediately by pressing the Play button. When should you select the automatically continue checkbox? You should select the automatically continue checkbox if you attach a breakpoint action to the breakpoint. When your program reaches the breakpoint, Xcode performs the action and goes back to running your program. Refer to the next section for information on breakpoint actions.

Breakpoint Actions

Breakpoint actions allow you to perform an action when reaching a breakpoint. To add a breakpoint action to a breakpoint, select the breakpoint, right-click, and choose Edit Breakpoint. Editing a breakpoint opens a pop-up editor for the breakpoint. Next to the Action label in the pop-up editor is a button with the text Add Action or Click to add an action. Click that button to add a breakpoint action. You can add the following breakpoint actions:

- Debugger command executes a GDB or LLDB command, depending on the debugger you're using.
- Log message writes a message to the debug console or speaks the message. Logging messages keeps you from having to add `NSLog` statements to your code and rebuilding your project.
- Sound plays a system sound.
- Shell command executes a Unix shell command. Use a shell command to run shell scripts when you reach a breakpoint.

- AppleScript executes an AppleScript script.
- Capture OpenGL ES frame captures the current OpenGL ES frame.

A breakpoint can have multiple breakpoint actions. Click the + button to create additional breakpoint actions.

Selecting the Automatically continue after evaluating checkbox tells Xcode to continue executing your program after performing the breakpoint action. If you want to view variables and step through your code, don't select the checkbox.

Debugger Command

Enter the command in the text field.

Log

Enter the message you want to log in the text view. If you want to log the value of a variable, wrap it in @ tags.

`@VariableName@`

Select the Speak message radio button to tell Xcode to speak the message instead of logging it to the debug console.

Sound

Playing a sound helps when debugging a fullscreen program. You can play a sound when reaching an area of your code. Use the pop-up menu to choose a sound.

Shell Command

Shell commands can be very powerful. You can even run command-line programs using shell commands. You could check for memory leaks when you reach a breakpoint by running the `leaks` command and supplying the name of your program.

If you add a shell command breakpoint action, two text fields appear in the pop-up editor. Enter the command in the first text field. If you've written a shell script that you want to use, click the Choose button. Enter the arguments the shell command takes in the second text field.

If the automatically continue checkbox is selected for the breakpoint, Xcode will not wait for the shell command to execute before continuing to run your program. Select the Wait until done checkbox to tell Xcode to wait for the shell command to finish before resuming execution of your program.

AppleScript

Enter your script in the text field. Use the Compile and Test buttons to make sure your script works before using it as a breakpoint action.

Capture OpenGL ES Frame

This action works only with iOS applications that use OpenGL ES. There is nothing to specify for the Capture OpenGL ES Frame action.

Sharing Breakpoints

Sharing a breakpoint lets other people access your breakpoints. Project and workspace breakpoints can be shared. To share a breakpoint select it in the breakpoint navigator, right-click, and choose Share Breakpoint. Selecting a file from the breakpoint navigator, right-clicking, and choosing Share Breakpoints shares all the breakpoints in the file.

Selecting a shared breakpoint, right-clicking, and choosing Unshare Breakpoint turns the shared breakpoint back to a normal breakpoint.

Launching the Debugger

To debug your project, activate breakpoints and run the project. Click the Breakpoints button on the project window toolbar to activate breakpoints. The button is dark gray when breakpoints are activated. Click the Run button to run your project in the debugger. If your program is currently running, choose Product > Attach to Process > AppName to debug the program.

When your project reaches a breakpoint, Xcode opens the debug area at the bottom of the project window. Figure 7.1 shows the debug area, which has three sections.

- Debug bar, which has buttons for the most common debugging commands.
- Variables view, which lets you examine the values of your program's variables.
- Console, which displays debugger commands along with the input and output of command-line programs.

On the right side of the debug area are three buttons that control the visibility of the variables view and console. The initial setup is to show both the variables view and console, but there are buttons that tell Xcode to show only the variables view and show only the console. Hiding the console helps if you're not logging any output to the console. Hiding the variables view helps if you feel more comfortable viewing your variables in the console.

Opening a Separate Console Window

If you write a lot of output to the debug console, you may want the console in a separate window so you can read all the output. Perform the following steps to create a separate console window:

1. Open the tab bar by choosing View > Show Tab Bar.
2. Add a new tab by clicking the + button on the right side of the tab bar.
3. Hide the navigator and utility area using the View group of buttons in the project window toolbar.
4. Hide the variables view using the buttons on the right side of the debug area.
5. Hide the editor by dragging the debug bar up, covering the editor.
6. Drag the tab off the tab bar.

Debug Bar

The debug bar provides a convenient way to perform common debugging tasks. On the left side of the debug bar are a series of buttons. The debug bar has the following buttons, running from left to right:

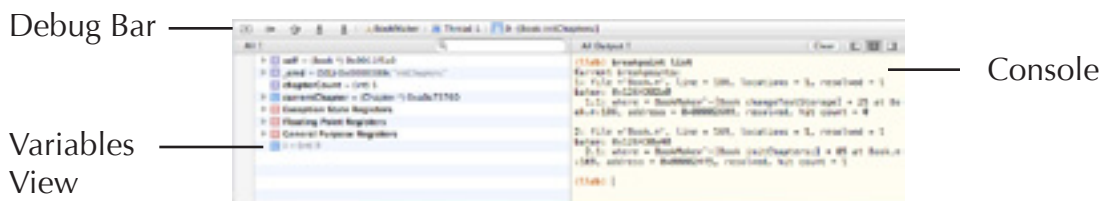


Figure 7.1

Debug area

- Show/Hide Debug Area
- Continue/Pause Execution
- Step Over
- Step Into
- Step Out

Read the section “Stepping Through Your Code” later in this chapter to learn more about the Step Over, Step Into, and Step Out buttons.

If you’re debugging an iOS application, you will see one or two additional buttons in the debug bar. The Simulate Location button lets you choose a city to simulate running your application. OpenGL ES applications have a Capture Frame button that captures the current OpenGL ES frame. Read the “OpenGL ES Debugging” section later in this chapter for more information on frame capture.

Next to the buttons on the debug bar is a jump bar. The jump bar lets you jump to any function on the call stack.

Debug Navigator

The debug navigator lets you view your program’s call stack, which is the chain of function calls that led your program to its current location. When your program reaches a breakpoint, the debug navigator should be visible on the left side of the project window. If the debug navigator is not visible, choose View > Navigators > Show Debug Navigator.

Selecting a function from the debug navigator takes you to its location in the editor, assuming you wrote that function. Sometimes low-level system calls appear in the call stack; selecting them shows assembly language code, which probably won’t help you much.

The debug navigator has a listing for each thread in your program. Your code will have at least two threads, one for your program and one for the debugger. In this case Xcode displays the call stack for your program’s main thread. If you write code with multiple threads, click the disclosure triangle next to a thread listing to view that thread’s call stack.

At the bottom of the debug navigator is a slider that controls how much of the call stack is visible in the debug navigator. Dragging the slider to the right shows more of the call stack; dragging the slider all the way to the right displays the entire call stack. Dragging the slider all the way to the left displays only the top of the call stack.

Next to the slider is a small button that looks like the symbol navigator icon in the navigator selector bar. Clicking the button shows only crashed threads and threads with debug symbols in the debug navigator. Showing only crashed threads and threads with debug symbols is a good way to display only your code’s call stacks in the debug navigator.

Xcode initially groups the call stacks by thread. Clicking the By Queue button at the top of the debug navigator tells Xcode to group the call stacks by dispatch queue. Xcode 4.4 replaces the By Thread and By Queue buttons at the top of the debug navigator with an icon button with an image of a spool of thread. Click the button to open a menu to group the call stacks by thread or by dispatch queue. Dispatch queues are part of Apple's Grand Central Dispatch technology, which allows your code to take advantage of multi-core systems.

The debug navigator also shows the memory locations you viewed in the memory browser. Read the "Viewing Memory" section later in this chapter for more information on the memory browser.

Floating Debugger Window

Xcode's project window takes up lots of screen space. When you're debugging, you may want the project window out of the way so you can concentrate on your application. Choose Product > Debug Workflow > Xcode Always In Front.

If you choose to have Xcode in front, Xcode creates a floating debugger window that is in front of all other applications. When your application is running, the floating debugger window shrinks to get out of the way. When you reach a breakpoint, the floating debugger window expands to show the debug navigator, editor, and debug area. Use the Debugging button in the floating debugger window toolbar to switch back to the standard window behavior. Stopping your application closes the floating debugger window and opens the project window.

On the right side of the floating debugger window's toolbar are two buttons: Focus and Console. The Focus button changes focus from your application to Xcode and vice versa. The Console button toggles showing the console in the floating debugger window. However, when I have used the floating debugger window, clicking the Console button did nothing.

Variables View

Use the variables view to look at your program's variables. The variables view displays the following information for each variable:

- An icon that identifies the type of variable.
- The variable name.
- The variable's data type, which is in parentheses. If you don't see a data type, right-click in the variable list and choose Show Types.

- The variable's value.
- A summary, which provides additional information about the variable. If you have a variable of type `NSString`, the summary tells you the string's contents. Because `NSString` variables are pointers, the string's value is its memory address. Many variables do not have a summary.

The icon that identifies the variable has a letter to identify the variable. Local variables have the letter L. Function arguments have the letter A. Instance variables, members of a class, have the letter I. Global variables have the letter G. Registers have the letter R.

Use the pop-up cell and search field above the variable list to control what variables appear in the variables view. The pop-up cell has three values: Auto, Local, and All Variables, Registers, Globals, and Statics. If you choose Auto, Xcode shows the most recently accessed variables. If you choose Local, Xcode shows local variables and function arguments. If you choose All Variables, Registers, Globals, and Statics, Xcode shows all variables.

The search field lets you filter the variables in the variables view. Suppose you're interested in only one variable. Enter the name of the variable in the search field, and Xcode hides all the other variables. You can also enter a value in the search field, and Xcode shows only the variables that have the entered value.

If a variable has multiple fields of information, it has a disclosure triangle next to it. Click the disclosure triangle to expand the variable so you can see the other fields. These fields may have additional fields; you can end up doing a lot of disclosure triangle clicking. Classes, data structures, and arrays are the variables most likely to have multiple fields.

Right-clicking a variable and choosing Edit Value lets you edit the value by typing in a new one. Suppose you want to test your program's handling of null pointers. By setting a pointer variable to `NULL` in the variables view, you force your null pointer handling code to execute.

Right-clicking a variable and choosing Print Description prints the description of the variable in the console.

Setting Watchpoints

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable. To set a watchpoint, select the variable from the variables view, right-click, and choose Watch "variable". If there is no Watch menu item, switch the debugger to GDB.

Custom Data Formatters

Custom data formatters let you customize what appears in the summary for each variable. Data formatters are especially useful for displaying the data structures you create for your programs. By using data formatters, you can display the most important data structure information in the variable's summary.

To create a custom data formatter, select a variable in the variables view, right-click, and choose Edit Summary Format. A pop-up editor opens. Enter a format string in the text field. Xcode lets you know if your format string is valid. Click the Done button when you're finished.

What does a format string look like? Any literal text you enter appears in the summary exactly as you type it, which makes the literal text good for labels. To refer to the variable itself in the format string, use the value `$VAR`. If the variable is a data structure, you can refer to the structure's individual members by placing the `%` character before and after the member name.

```
%MemberName%
```

If one of your data structure's members is another data structure, use the dot operator. Suppose you have a data structure with a member named `position`, which in turn has members `x` and `y`. You want to show the position's `x` component in the summary. You would type the following for the position's format string:

```
%position.x%
```

Let's walk through a simple example of using data formatters. Suppose you have a data structure for 3D vectors named `Vector3D` with members `x`, `y`, and `z`. You would like the summary to show the `x`, `y`, and `z` components of the vector. Select the `Vector3D` variable in the variables view, right-click, and choose Edit Summary Format. Enter the following text in the Set Summary Format text field:

```
x=%x%, y=%y%, z=%z%
```

Now every `Vector3D` variable in your program should show the `x`, `y`, and `z` components in the summary. If `x` has a value of 1, `y` has a value of 3, and `z` has a value of 5, the variable's summary shows the following output:

```
x=1, y=3, z=5
```

Data formatter strings are stored in the following location:

```
/Users/YourUsername/Library/Developer/Xcode/UserData/  
Debugger/CustomDataFormatters
```

Datatips

Datatips provide debugging information for a variable in an Xcode editor. They let you see a variable's value without having the debug area open.

Using Datatips

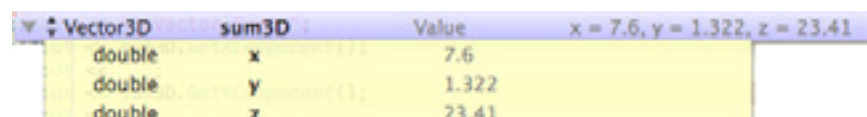
Moving your mouse over a variable in the editor displays basic information about the variable: its name, data type, value, and any summary information. Figure 7.2 shows an example of a datatip. If the variable has additional information, there will be a disclosure triangle on the left side. Move the mouse over the disclosure triangle to see the additional information. Classes, data structures, and arrays are the variables most likely to have additional information.

When the information on a variable is visible in the editor and you move the mouse cursor over the variable, a pop-up button cell (it looks like a tiny set of up/down arrows) appears next to the variable. Click the cell to open a menu. Use the menu to print a description of the variable and view the variable in the memory browser.

Clicking a variable's value lets you change its value.

Using Step Controls in the Editor

Moving the mouse in the gutter next to a line of code opens a step control. If you're at the current line of code, the step control is a Step Over button. For other lines of code, the step control is a Continue button that tells the debugger to continue running until it reaches the line of code where you clicked the Continue button.



Vector3D		sum3D	Value	x = 7.6, y = 1.322, z = 23.41
double	x		7.6	
double	y		1.322	
double	z		23.41	

Figure 7.2

Datatip example

Viewing Shared Libraries

Choose Product > Debug > Shared Libraries to open the shared libraries window. The shared libraries window lists all the loaded libraries for your program. The window shows the following information for each shared library:

- Name, which is the name of the library.
- Address, which is where the library resides in memory.
- Path, which is the path to the library in the file system.
- Debug Symbols, which tells you if the library's debug symbols are loaded. If the debug symbols have not been loaded, the Debug Symbols column has a Load button you can click to load the symbols.

At the bottom of the shared libraries window is a search field to filter the libraries that appear in the window. Click the Done button to close the shared libraries window.

Showing debugging symbols strikes a balance between the ability to debug and speed of debugging. Mac OS X and iOS applications use a lot of shared libraries, and these libraries have lots of symbols. Loading every one of these symbols takes time and makes debugging slower. Load only the debug symbols you need.

Tracking Expressions

An expression is a piece of code that returns a value. The most common expressions are variables and the results of functions and methods. To track an expression, right-click in the variables view and choose Add Expression. A pop-up editor opens. Enter the expression in the text field. Select the Show In All Stack Frames checkbox if you want to see the expression in all stack frames. Click the Done button to finish adding the expression. If you're tracking a function, cast the function to its return type.

(ReturnType)FunctionName

Make sure you type the expression correctly. If you misspell the expression or enter a non-existent variable name, Xcode won't tell you about the error, except to provide the summary value of "invalid expression" in the variables view. If you make a mistake, select the expression, right-click, and choose Edit Expression. An expression has an icon with the letter E next to it in the variables view.

To stop tracking an expression, select it from the variables view and press the Delete key.

Viewing Dynamic Arrays

Many programs use pointers to create arrays when the size cannot be determined until the program runs. To create these dynamic arrays, you declare a pointer variable, then use the `malloc()` (in C) or `new()` (in C++) calls to make the pointer large enough to store the array. When you look at the pointer variable, you want to look at the array, not the variable. If you look at the pointer variable in the variables view, you'll see only the pointer, not the array. How do you view the array?

Create an expression for the array. Right-click in the variables view and choose Add Expression. A pop-up editor opens. Enter the expression in the text field and click the Done button. The expression should take the following form:

```
*ArrayName@Length
```

If you want to see the first 25 elements of the array `myArray`, enter the following expression in the text field:

```
*myArray@25
```

Stepping Through Your Code

When debugging you will spend a lot of time stepping through your code. Stepping means executing one line of code at a time, letting you pinpoint problems in your program.

The debug area's jump bar has three buttons for stepping: Step Over, Step Into, and Step Out. To demonstrate stepping I will use a dummy C function.

```
void ExampleFunction(void)
{
1   int routinesCalled;

2   routinesCalled = 0;
3   FirstRoutine();
4   routinesCalled++;

5   SecondRoutine();
6   routinesCalled++;
}
```

Assume you've paused the program at `ExampleFunction()`. The debugger moves past line 1 and stops at line 2. For this line of code, the Step Over and Step Into buttons do the same thing. They execute the line of code, giving `routinesCalled` the value zero, and move to line 3.

If you click the Step Over button again, the debugger executes all the code in the function `FirstRoutine()` and moves to line 4. The debugger steps over the function call and moves to the next line of code. Clicking Step Over a second time moves the debugger to line 5.

Clicking the Step Into button at this point moves the debugger inside `SecondRoutine()`. From there you can step through the code in `SecondRoutine()`, which I have not bothered to list. The debugger steps into the function you're calling. After executing all the lines of code in `SecondRoutine()`, the debugger moves to line 6. If you don't want to walk through every line of code in `SecondRoutine()`, click the Step Out button when you're finished looking at `SecondRoutine()`. The debugger will step out of `SecondRoutine()` and take you back to `ExampleFunction()`.

On a line of code that calls another function, clicking Step Over executes the code in the function. Clicking Step Into takes you inside the called function. Clicking Step Out takes you out of the called function.

Viewing Memory

The memory browser lets you examine the contents of memory. To open the memory browser, right-click a variable in the variables view and choose View Memory of VariableName. If the variable is a pointer, you can either view the memory of the pointer or the memory of the variable the pointer references.

At the bottom of the memory browser, shown in Figure 7.3, are controls to customize what appears in the memory browser and control how the memory appears. The Address text field lets you choose the starting memory address to appear in the browser. You can enter an address in the text field or use the Memory Page buttons to choose the starting address.

The Memory Page buttons step through memory. The amount to step is determined by the value of the Number of Bytes pop-up menu. If the Number of Bytes menu has the value of 512 bytes, clicking the Memory Page buttons moves up or down 512 bytes from the current address in the browser.

The Number of Bytes pop-up menu determines the amount of memory the browser displays at one time. The memory browser can display 64 to 65536 bytes.

The Byte Grouping pop-up menu determines how many bytes of memory are grouped into one word in the memory browser. The possible byte groupings are 1, 2, 4, 8, 16, and 32 bytes. If you choose None from the menu, you get a one-byte grouping, which means the memory browser displays the memory as a series of two-digit hexadecimal numbers. A two-byte grouping makes the memory browser display the memory as a series of four-digit hex numbers. A 32-byte grouping makes the memory browser display the memory as a series of 64-digit hex numbers.

Each row in the memory browser displays the following information:

- The starting address for the row.
- The contents of memory, displayed as hexadecimal numbers.
- The ASCII character corresponding to the value of each byte of memory in the row.

Clicking the Lock button prevents the view of the memory from updating. Normally the memory updates when you step over a line of code, even if the contents of the memory don't change. By clicking the Lock button you save the view of the memory so you can analyze it more deeply.

The jump bar lets you go to a previously viewed memory address.

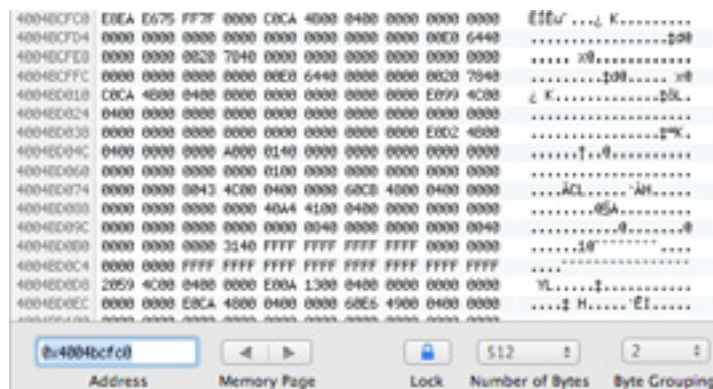


Figure 7.3

Memory browser

OpenGL ES Debugging

Xcode has the ability to capture the current OpenGL ES frame. You must be running iOS 5 or later on your device to be able to capture the frame.

Enabling OpenGL ES Frame Capture

If you are running Xcode 4.4 or later, OpenGL ES frame capture is automatically enabled. If you are running an older version of Xcode, you must enable OpenGL ES frame capture in your project. Perform the following steps to enable OpenGL ES frame capture:

1. Choose Edit Scheme from the Scheme menu in the project window toolbar.
2. Select the Run action from the left side of the scheme editor.
3. Click the Options button in the scheme editor.
4. In Xcode 4.4 the OpenGL ES Frame Capture menu should be set to Automatically Enabled. If you are running an older version of Xcode, select the Enable frame capture checkbox.

Capturing the Frame When Reaching a Breakpoint

You can tell Xcode to capture the frame when you reach a breakpoint by using the OpenGL ES frame capture breakpoint action. Perform the following steps to add the breakpoint action:

1. Right-click a breakpoint in the gutter or the breakpoint navigator.
2. Choose Edit Breakpoint to open a pop-up editor.
3. Click the Add Action button.
4. Choose Capture OpenGL ES Frame from the Action menu.

Capturing the Frame

Click the Capture Frame button in the debug bar to capture the current frame. The Capture Frame button is next to the Choose Location button. You can also choose Product > Debug > Capture OpenGL ES Frame. Click the Release Frame button (it looks like the Continue button) when you're finished viewing the frame capture data to go back to normal debugging.

Framebuffer Area

When you capture a frame, Xcode displays the framebuffer above the debug area, where the editor usually is. Figure 7.4 shows an example of the framebuffer area. Below the framebuffer are three groups of buttons: Buffers, View, and Orientation. The Buffers group has buttons to show the color, depth, and stencil buffers in the framebuffer area. Clicking the Auto button tells Xcode to show all the buffers you're currently using.

The View group has buttons to zoom the buffers. The Orientation group has buttons to rotate the buffers.

The framebuffer area has a button in the lower left corner and a button in the lower right corner. Clicking the button in the lower left corner opens a pop-up editor that tells you the size and color format of the renderbuffer. Clicking the button in the lower right corner opens a pop-up editor that lets you control the color channels that appear in the framebuffer. Click the color to enable that color channel. Use the slider to control the channel values that appear in the framebuffer.

The right side of the debug bar (refer to Figure 7.5) has controls that let you step through OpenGL ES draw calls. The Back and Forward buttons move you through each call. The objects that were drawn in that draw call have a green wireframe highlight. The color buffer for that call also appears on the device. You can also use the slider to step through the draw calls.



Figure 7.4

Framebuffer area



Figure 7.5

Debug bar in frame capture mode.

Debug Navigator

When debugging an OpenGL ES application, Xcode displays the OpenGL ES commands your application made in the debug navigator. The commands are listed in the order your application made them. Clicking the disclosure triangle next to a command shows the call stack at the time the command was called. You can also choose an OpenGL ES call from the jump bar.

You can add debug markers to the debug navigator using Apple's `APPLE_debug_marker` OpenGL ES extension. Call the functions `glPushGroupMarkerEXT()` and `glPopGroupMarkerEXT()` in your code to add debug markers. When calling `glPushGroupMarkerEXT()`, supply a length and a name. The length can be zero. Debug markers show up as folders in the debug navigator. The following code demonstrates adding debug markers:

```
glPushGroupMarkerEXT(0, "Marker Name");

// OpenGL ES calls you want to annotate

glPopGroupMarkerEXT();
```

Variables View

When you're in frame capture mode, the variables view shows OpenGL ES state information, which you can see in Figure 7.6. There are two columns of information in the debug area with no console. Each column has a menu above the information that lets you control the state information the column displays. There are four groups of state information you can display: GL Context, Bound GL Objects, All GL Objects, and Context Info.

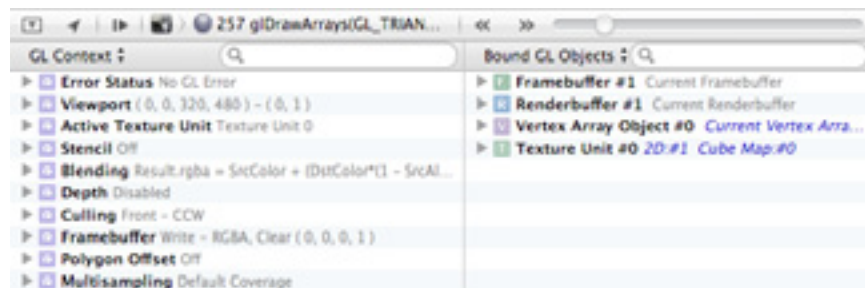


Figure 7.6

Debug area in frame capture mode

Choosing GL Context shows state information, such as the buffers used, the size of the viewport, and the active texture unit. Choosing Bound GL Objects shows the bound objects, such as framebuffers and texture units. Choosing All GL Objects shows both bound and unbound objects. Choosing Context Info shows information about the draw context, including the name of the renderer, the OpenGL ES extensions the context supports, and the context's limits. An example of a limit is the maximum texture size.

Assistant Editor

Opening the assistant editor allows you to see the OpenGL ES objects associated with the captured frame. The top level of the jump bar has the following categories: Bound Objects, All Objects, and Stack. Initially the assistant editor shows all bound objects. Choosing All Objects shows both bound and unbound objects. Choosing Stack allows you to view call stack functions in the assistant editor.

If you decide to show bound objects or all objects, choosing All from the jump bar's submenu shows every object. You can also use the jump bar's submenu to show a subset of objects, such as showing only textures. Double-clicking an object in the assistant editor provides a detailed view of the object. For a graphical object like a texture, Xcode shows the same information it shows for a buffer: an image, the View group of buttons, the Orientation group of buttons, and the Info and Color Channel buttons. Clicking the Info button for a texture tells you the texture's size, format, filtering parameters, and wrapping parameters. For an object that contains vertex data like a VAO (vertex array object), Xcode unpacks the vertex data in the editor so you can examine it.

When you choose a call stack function from the jump bar, Xcode opens an editor where you can view the source code for that function. By opening an editor you can view your source code as you step through OpenGL ES calls. If you have the editor open, selecting an OpenGL ES call from the debug navigator shows the source code in the editor and takes you to the OpenGL ES call in the editor.

Labeling OpenGL ES Objects in the Debugger

Apple has an OpenGL ES extension, `APPLE_debug_label`, that allows you to label OpenGL ES objects so you can identify them in the debugger. Call the function `glLabelObjectEXT()` in your source code to add a label. The function takes the following form:

```
glLabelObjectEXT(objectType, objectName, length, labelName);
```

The following code labels a background texture:

```
glLabelObjectEXT(GL_TEXTURE, backgroundTexture, 0,  
    "Background");
```

Using the GDB Console

The Xcode debugger handles basic debugging tasks; you can step through your code, examine the values of variables, and set breakpoints. Many of you will be able to do all your debugging from the Xcode GUI. But if you want to do anything more advanced, you must use the console and type commands in it. Xcode uses two debuggers for C, C++, and Objective-C programs: GDB and LLDB. This section covers the GDB debugger.

Using the console doesn't mean you have to abandon the Xcode GUI. You can use the debug bar to step through your code and view your program's variables in the variables view while using the console to perform tasks the Xcode GUI can't handle.

Before you can type commands in the console, you must pause your program's execution in the debugger. If you've set breakpoints in your program, you can wait for your program to reach one of them. Otherwise you can click the Pause button in the debug bar to start entering commands in the console.

Stopping Program Execution

There are three ways to stop your program's execution without explicitly pausing your program: breakpoints, watchpoints, and catchpoints. As I mentioned earlier in this chapter, breakpoints stop your program's execution at a specific line of code in your program. You can set breakpoints in the GDB console the way you would from the Xcode GUI, but GDB provides even more options. From the console you can set conditional breakpoints and set breakpoints that execute only one time.

Watchpoints stop your program's execution when an expression changes value. Most of the time the expression you're interested in is a variable.

Catchpoints stop your program's execution when a C++ exception occurs. Exceptions provide a way of handling run-time errors in C++ programs. The C++ statements `try`, `catch`, and `throw` deal with exceptions. Obviously, only C++ programs can use catchpoints.

Setting Breakpoints

The `break` command, which you can abbreviate by typing `b`, sets breakpoints in the console. There are many ways to set them in GDB. If you supply no arguments,

```
break
```

GDB sets a breakpoint at the current line of code. Supplying a line number tells GDB to set a breakpoint at that line number in the current source code file. The following command:

```
break 447
```

Tells GDB to set a breakpoint at line 447 in the current file. To set a breakpoint at a specific line in a source code file, supply the file name, a colon, and the line number. The following command:

```
break main.m:15
```

Tells GDB to set a breakpoint at line 15 in the file `main.m`. Rather than setting a breakpoint at a line of code, you may prefer to set a breakpoint at the start of a function. Supply the function name when calling `break`. If you have a C function `PlayMusic()` that you want to set a breakpoint at, you would enter the following command:

```
break PlayMusic
```

Object-oriented languages like C++ and Objective-C complicate matters slightly. You must also supply a class name along with the function name because multiple classes can have a function with the same name. If you have a C++ class named `TMusic` with a member function `Play()`, you would enter the following command:

```
break TMusic::Play
```

If you have an Objective-C class named `TMusic` with a `play:` method, you would enter the following command:

```
break -[TMusic play:]
```

To set a breakpoint that stops your program only once, call `tbreak`. The `t` in `tbreak` stands for temporary. The `tbreak` command takes the same arguments as `break` does. The following command:

```
tbreak main.m:15
```

Sets a breakpoint at line 15 in the file `main.m`, then deletes the breakpoint when your program stops at it.

Setting Watchpoints

The `watch` command sets watchpoints in your program. You supply an expression, which in most cases is a variable name. When the expression's value changes, GDB pauses your program.

```
watch VariableName
```

Setting Catchpoints

The `catch` command sets catchpoints in your program. There are two ways to call `catch`.

```
catch catch  
catch throw
```

The first statement stops your program when it catches a C++ exception, and the second statement stops your program when it throws a C++ exception.

Examining Your Breakpoints

To view a list of all the breakpoints, watchpoints, and catchpoints you set, call the `info breakpoints` command. For each breakpoint, watchpoint, and catchpoint, GDB displays the information shown in Table 7.2.

Setting Conditional Breakpoints

Conditional breakpoints give you the power to control when a breakpoint pauses your program. Supply a condition when you set the breakpoint. When your program hits the line of code where you set the conditional breakpoint, GDB pauses the program if the condition you supply is true.

There are two ways to set conditional breakpoints. First, supply a condition when you create the breakpoint with the `break if` command. The condition can be any Boolean expression, an expression that can be either true or false. Suppose you have the following code in a program:

Table 7.2 Information info breakpoints Provides

Information	Description
Breakpoint number	Breakpoints start at 1. Many breakpoint-related GDB commands take breakpoint numbers as parameters.
Type	Tells you whether you have a breakpoint, watchpoint, or catchpoint.
Disposition	Specifies what should happen when your program hits the breakpoint. The value <code>keep</code> tells GDB to keep the breakpoint. The value <code>dis</code> tells GDB to disable the breakpoint. The value <code>del</code> tells GDB to delete the breakpoint.
Enabled	Is the breakpoint enabled? If so, it will have the value <code>y</code> . If not, it will have the value <code>n</code> .
Address	The breakpoint's memory address.
What	The function name, source code file, and line number of the breakpoint.
Condition	If the breakpoint is conditional, the condition will reside here. If the breakpoint has been hit before, the number of times it's been hit will be here as well.

```
enum DirectionType {
    NORTH, SOUTH, EAST, WEST
};

void Move(DirectionType directionToMove);
```

The following breakpoint executes when the program moves north:

```
break Move if directionToMove == NORTH
```

You can also use `if` to set conditional watchpoints, but you won't use conditional watchpoints as much as conditional breakpoints. Watchpoints pause your program when an expression changes value. The changing of value is what you're normally interested in so regular watchpoints are usually sufficient. Use a conditional watchpoint to pause your program when an expression changes to an interesting value.

```
watch x if x > 50
```

Second, use the `condition` command to turn a regular breakpoint into a conditional one. With the `condition` command you can set your breakpoints in Xcode and use the console to make certain ones conditional. Supply a breakpoint number and a condition with the

`condition` command. The following example shows how you would use the `condition` command to pause when the `Move()` function moves north, assuming the breakpoint number is 1:

```
condition 1 directionToMove == NORTH
```

You can also use the `condition` command to make watchpoints and catchpoints conditional. Use conditional catchpoints to pause your program when it raises a specific exception.

To make a conditional breakpoint, watchpoint, or catchpoint unconditional, use the `condition` command. Supply a breakpoint number and no condition, and GDB removes the condition from the breakpoint, as you can see in the following example:

```
condition 1
```

Closely related to conditional breakpoints is GDB's ignore count. The ignore count lets you skip a breakpoint when your program reaches it. Normally the ignore count is zero, meaning you won't skip any breakpoints, but you can use the `ignore` command to ignore a breakpoint. Supply a breakpoint number and a desired ignore count, and GDB will ignore that breakpoint the number of times you specify. The following example skips breakpoint 2 seven times:

```
ignore 2 7
```

When you reach a breakpoint in your program, you can set the ignore count for that breakpoint when you resume if you use the `continue` command from the console instead of clicking the Continue button in the Xcode debug area. The following example sets the ignore count to nine:

```
continue 9
```

Disabling and Deleting Breakpoints

Let's say you set a breakpoint inside a loop. If the loop runs hundreds of times, you may not want to step through your code every time through the loop. Disabling the breakpoint turns it off but keeps it in the breakpoints list so you can turn it back on later.

The `disable` command disables breakpoints, watchpoints, and catchpoints. Supply a breakpoint number or a range of breakpoint numbers. The `info breakpoints` command shows your program's breakpoint numbers.

```
disable 4
disable 3-6
```

To enable disabled breakpoints, watchpoints, and catchpoints, use the `enable` command. Like the `disable` command, you supply either a breakpoint number or a range of numbers.

```
enable 2
enable 8-13
```

Calling `enable` like I did in the previous example causes the breakpoint to execute every time your program hits the breakpoint. Calling `enable` with the `once` option disables the breakpoint after it stops your program once.

```
enable once 1
enable once 5-6
```

Calling `enable` with the `delete` option deletes the breakpoint after it stops your program once.

```
enable delete 9
enable delete 2-7
```

To delete a breakpoint, watchpoint, or catchpoint, use the `delete` command. Like the `enable` and `disable` commands, you can supply a breakpoint number or a range of numbers.

```
delete 16
delete 11-14
```

If you want to delete the breakpoint you've just reached, use the `clear` command. Supply no arguments to delete the breakpoint you reached. You can also supply a line number or a function where you want to delete the breakpoint.

```
clear                // Clear the current breakpoint
clear main.m:15      // Clear a breakpoint at a line
clear PlayMusic      // Clear a C function breakpoint
clear TMusic::Play   // Clear a C++ member function
                    // breakpoint
clear -[TMusic play:] // Clear an Objective-C method
                    // breakpoint
```

Command Lists

Command lists allow you to run a series of commands when the debugger hits a breakpoint, watchpoint, or catchpoint. A command list adheres to the following format:

```
commands [Breakpoint Number]
    Insert the commands you want to execute here
end
```

If you create the command list immediately after creating the breakpoint (or watchpoint or catchpoint), you do not have to supply a breakpoint number; GDB attaches the command list to the breakpoint you just created. Use the `info breakpoints` command to get the numbers of all your program's breakpoints, watchpoints, and catchpoints.

When you start building a command list by typing commands, the prompt changes from (gdb) to > and will not revert back to (gdb) until you finish building the list by typing `end`. Many command lists have `silent` as the first command. The `silent` command suppresses the printing of the message that you hit a breakpoint, which helps if your command list prints information to the console. For the `silent` command to work, it must be the first command in a command list.

You can execute as many commands as you want in a command list, and you can use any GDB command in the list. The following command list writes the contents of an integer variable `y` to the console and continues the program's execution when it reaches the first breakpoint in the program:

```
commands 1
    silent
    printf "y = %d \n", y
    continue
end
```

You can also specify a command list when you create a breakpoint. The following example demonstrates creating a command list after creating a conditional breakpoint:

```
break main.m:24 if y > 5
commands
    silent
    printf "y = %d \n", y
    continue
end
```

NOTE

You don't have to indent the commands in the GDB console. I indented the examples to make them easier to read.

Examining Data

Xcode's variables view lets you view your variables and their values. The memory browser lets you view the contents of memory. From the GDB console you can view dynamic arrays. You can also tell the console to show the values of variables automatically when your program stops execution.

Examining Dynamic Arrays

Many programs use pointers to create arrays when the size cannot be determined until the program runs. To create these dynamic arrays, you declare a pointer variable, then use the `malloc()` (in C) or `new()` (in C++) calls to make the pointer large enough to store the array. When you look at the pointer variable, you want to look at the array, not the variable. If you look at the pointer variable in the variables view of Xcode's debug area, you'll see only the pointer, not the array.

Artificial arrays solve the pointer problem. The `@` operator defines the array. The array goes to the left of the `@` operator, and the length of the array goes to the right of the array. Use the `print` command to look at the contents of the array.

```
print *array@length
```

The length is the number of elements you want to display. Suppose you have a 1000 element array, `myArray`, and you want to look at the first 50 elements. You would view the 50 elements with the following command:

```
print *myArray@50
```

If you want to start looking from a position other than the start of the array, use the following notation:

```
print *(array + starting point)@length
```

If you want to skip the first 100 elements and look at the next 25, you would run the following command:

```
print *(myArray + 100)@25
```


Displaying Data Automatically

When you're debugging, there's a few variables you're especially interested in. GDB allows you to place these variables in a display list, which GDB displays every time your program stops. Display lists keep you from having to click a series of disclosure triangles in the variables view to look at an important variable.

To add a variable to the display list, use the `display` command. Supply a variable name, and GDB adds it to the list.

```
display VariableName
```

You can also supply any of the viewing formats in Table 7.3 to tell GDB to display the data the way you want. The following example displays a variable as a floating-point number:

```
display/f y
```

To see a list of all the variables you're automatically displaying, use the `info display` command. It tells you the following pieces of information for each variable:

- The ID number, which you can use to turn off automatic display.
- Whether or not it's enabled, y for yes and n for no.
- The expression to display.

To temporarily disable a variable from the automatic display list, use the `disable display` command. Supply ID numbers (separated by spaces) to disable, which you can see in the following examples:

Table 7.3 Data Viewing Formats

Format	Description
x	Display as a hexadecimal (base 16) integer. This is the default for viewing memory.
d	Display as a signed decimal integer.
u	Display as an unsigned decimal integer.
o	Display as an octal (base 8) integer.
t	Display as a binary (base 2) integer.
a	Display as hexadecimal address and as an offset from the nearest symbol.
c	Display as a character constant.
f	Display as a floating-point number.

```
disable display 5          // Disable one variable
disable display 1 3 6      // Disable multiple variables
```

The `enable display` command enables variables you disabled with the `disable display` command. Supply ID numbers to enable. The following examples enable the variables I disabled in the previous example:

```
enable display 5
enable display 1 3 6
```

To permanently remove variables from the automatic display list, use either the `undisplay` or `delete display` commands. Like the `enable display` and `disable display` commands, you supply a list of ID numbers separated by spaces, which you can see in the following examples:

```
undisplay 3
undisplay 2 4 5 9
delete display 7
delete display 2 6 13
```

You can place memory locations in automatic display lists so you can look at important memory addresses every time your program stops. Use the `display` command, supplying the memory address. Table 7.4 lists the memory units you can use to display memory. The following are some examples of using display lists:

```
display/64b myPtrVariable // Display 64 bytes automatically
display/16i 0x23459900    // Display 16 machine instructions
display/100fg 0x12123400  // 100 giant words as floating-point
```

Table 7.4 Memory Units

Unit	Letter You Pass to GDB	Bytes
Bytes	b	1
Halfwords	h	2
Words	w	4
Giant Words	g	8
Machine Instructions	i	N/A
String	s	N/A

Executing Shell Commands

The `shell` command lets you execute a Unix shell command in GDB. It takes the following form:

```
shell CommandName
```

With the `shell` command you can run another program in the GDB console. The following command runs the `heap` program that shows your program's memory heap:

```
shell heap AppName
```

The `shell` command also lets you run shell scripts, giving you enormous power if you know how to write Unix shell scripts. You can also place shell commands in a command list to execute scripts of shell commands when your program hits a breakpoint.

Defining Your Own Commands

If you find yourself entering a sequence of commands repeatedly, define a command to save yourself some typing. A user-defined command consists of a series of GDB commands. User-defined commands are similar to command lists. The difference is command lists execute when your program reaches a breakpoint while user-defined commands execute when you submit the command.

The first command in a user-defined command is the `define` command, which gives the command a name. The `define` command takes the following form:

```
define CommandName
```

After entering the `define` command, the console prompts you to enter the GDB commands you want to place in the user-defined command. A user-defined command can have up to ten arguments, with the first argument having the name `$arg0` and the tenth argument having the name `$arg9`. The `end` command is the equivalent of `}` in a C or Java program. Use the `end` command to end your user-defined command and return the prompt in the console to normal.

Let's look at an example of a simple user-defined command. Suppose you want to create a command that prints the contents of a dynamic array. You would define the following command:

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end
```

In this example `$arg0` is the pointer to the array, `$arg1` is the first element you want to display, and `$arg2` is the number of elements to display. To display the first 20 elements of an array called `myArray`, you would call the `PrintArray` command you created.

```
PrintArray myArray 0 20
```

GDB cannot alert you to errors in your user-defined command as you're entering the GDB commands. Running the command is the only way to know you wrote the command correctly. You have the responsibility of making sure you entered the GDB commands correctly.

Conditional Commands

You may want some of the GDB commands in your user-defined command to execute multiple times and other commands to execute conditionally. GDB has `if` and `while` commands that work similarly to `if` and `while` statements in C. Use the `if` and `while` commands to create conditional commands in your user-defined command.

The `if` command executes a series of GDB commands if a condition is true. It takes the following form:

```
if Condition
    Commands
else
    Commands
end
```

To make the `PrintArray` command example check for a negative starting point and a negative number of items, you would use the following command:

```
define PrintArray
    if ($arg1 < 0) || ($arg2 < 0)
        printf "ERROR! \n"
    else
        print *($arg0 + $arg1)@$arg2
    end
end
```

The `while` command executes a series of GDB commands repeatedly until a condition is false. It takes the following form:

```
while Condition
    Commands
end
```

The following example prints the first `x` elements of a static array one element at a time, where `x` is a number you supply:

```
define PrintStaticArray
    set $array = $arg0
    set $index = 0
    while ($index < $arg1)
        print $array[$index]
        printf("\n")
        set $index = $index + 1
    end
end
```

The variables `$array` and `$index` are convenience variables. Convenience variables store values in GDB so you can use the values later in the debugging session. They have no effect on your program. In the `PrintStaticArray` example the `$index` convenience variable keeps track of where GDB currently is in the array. Convenience variables start with `$`. The name of the convenience variable cannot be a number.

```
$3 // Invalid convenience variable name
```

GDB uses variable names with numbers as a value history of variables you print to the console. When you print a variable, GDB creates a value history variable and displays the value history variable in the console. You can refer to value history variables in other GDB commands.

Documenting Your Commands

If you create commands for other programmers to use, you want to document the commands so the other programmers know what the command does. The `show user` command displays the GDB commands that make up a user-defined command. If you supply no arguments, the `show user` command displays the GDB commands for all user-defined commands. Supply a command name to show the GDB commands for a single user-defined command.

```
show user CommandName
```

To create more documentation about a command, use the `document` command. It takes the following form:

```
document CommandName
```

After executing the `document` command, the console prompts you to enter the documentation. Press the Return key to end a paragraph of documentation. When you're finished entering the documentation, press the Return key and type `end`. Now when someone runs `help` on your command, the documentation you entered appears in the console.

Reading Commands from a File

Defining your own commands is a powerful tool, but there are problems if you define commands you want to use over a long period of time. Entering the commands every time you launch GDB becomes annoying after a while. Plus, you could make a typographical error, forcing you to reenter the command.

The solution is to place the commands you want to define in a file and load the file when you launch GDB. Place one GDB command per line. The character `#` signifies a comment. GDB ignores blank lines and tabs so you can indent commands to make them easier to read. You can also enter documentation for your commands in the file. Make sure to define the command first. The following example shows how the `PrintArray` command would look in a file:

```
define PrintArray
    print *($arg0 + $arg1)@$arg2
end
```

```
document PrintArray
```

```
The PrintArray command displays the contents of dynamic
arrays. It takes three arguments.
```

```
$arg0  The array.
$arg1  First element to display.
$arg2  Number of elements to display.
```

```
end
```

The `source` command executes commands from a file. Supply a file name.

```
source Filename
```

In the case of user-defined commands, running the `source` command loads the commands and documentation from the file. After loading the commands from the file, you can execute them when needed.

To have your user-defined commands loaded when GDB starts, place the commands in a file and perform the following steps:

1. Move the file you created to your home directory, `/Users/YourUsername`.
2. Run the Terminal program to reach the command line.
3. Change the name of your file to `.gdbinit` from the command line using the `mv` command.

The Finder won't let you start a file's name with a period because the operating system allows only system files to begin with a period. To start a file's name with a period, you must change the file's name from the command line.

When GDB launches it loads the `.gdbinit` file and executes any commands in the file. In the case of user-defined commands, GDB defines the commands on startup so you can start using your commands immediately.

Command Hooks

A *command hook* lets you execute a series of GDB commands before or after a command executes. You can create command hooks for every GDB command as well as user-defined commands. A command hook has the same structure as a user-defined command.

```
define
    Commands in hook
end
```

When defining a command hook to execute before the command, the `define` command takes the following form:

```
define hook-CommandName
```

When defining a command hook to execute after the command, the `define` command takes the following form:

```
define hookpost-CommandName
```

The following example prints a header for the `PrintArray` command example:

```
define hook-PrintArray
    printf "Array Contents\n\n"
end
```

Using the LLDB Console

The Xcode debugger handles basic debugging tasks; you can step through your code, examine the values of variables, and set breakpoints. Many of you will be able to do all your debugging from the Xcode GUI. But if you want to do anything more advanced, you must use the console and type commands in it.

Using the console doesn't mean you have to abandon the Xcode GUI. You can use the debug bar to step through your code and view your program's variables in the variables view while using the console to perform tasks the Xcode GUI can't handle.

Before you can type commands in the console, you must pause your program's execution in the debugger. If you've set breakpoints in your program, you can wait for your program to reach one of them. Otherwise you can click the Pause button in the debug bar to start entering commands in the console.

NOTE

New versions of Xcode incorporate updates to LLDB. If you are running an older version of Xcode, you may not have access to some commands and options I cover in this section.

Getting Help

If you look in the Organizer for LLDB documentation, you will be disappointed. Currently LLDB documentation is available only from the console, either using Xcode's debug console or launching LLDB from the Terminal. Enter the `help` command to get help. If you supply no arguments to the `help` command, it displays a list of debugger commands and command abbreviations. Supply a command category to get a list of commands available for that category. The following command shows the breakpoint-related commands available in LLDB:

```
help breakpoint
```

Supply a command name to get help for a specific command. The following command shows help on the `breakpoint modify` command:

```
help breakpoint modify
```


Setting Breakpoints

Use the `breakpoint set` command to set a breakpoint in the LLDB console. There are many ways to set a breakpoint in the console. Using the `-l` flag and entering a line number tells the debugger to set a breakpoint at that line number in the current file. The following command tells LLDB to set a breakpoint at line 134 in the current file:

```
breakpoint set -l 134
```

To add a breakpoint at a specific file, use the `-f` option and supply a file name. The following command tells LLDB to set a breakpoint at line 50 in the file `MyDocument.m`:

```
breakpoint set -f MyDocument.m -l 50
```

The easiest way to set a symbolic breakpoint is to use the `-r` option and enter the function name. The `-r` option works for C, C++, and Objective-C code. Using the `-r` option is the easiest option for Objective-C methods because all you have to enter is the method name. The following command sets a symbolic breakpoint in the Objective-C method `readFromFileWrapper`:

```
breakpoint set -r readFromFileWrapper
```

C++ programs can also use the `-M` option to set a symbolic breakpoint in a C++ method. Enter the method name. The following command sets a breakpoint in a C++ game's `GameLoop()` method:

```
breakpoint set -M GameLoop
```

If Xcode is unable to create a symbolic breakpoint, it displays the following message in the console:

```
WARNING: Unable to resolve breakpoint to any actual
locations.
```

Setting Watchpoints

The `watchpoint set` command sets watchpoints. You can set watchpoints on expressions or variables. Setting a watchpoint on an expression takes the following form:

```
watchpoint set expression Expression
```

Setting a watchpoint on a variable takes the following form:

```
watchpoint set variable VariableName
```

Examining Breakpoints

Enter the command `breakpoint list` to see the breakpoints you set. LLDB displays the following high-level information for each breakpoint:

- The breakpoint number.
- The type of breakpoint: file, symbolic, or regular expression. Breakpoints set with the `-r` option are regular expression breakpoints.
- The line number for file breakpoints.
- Locations, which is the number of breakpoint locations.
- Resolved, which is the number of resolved locations. An unresolved breakpoint occurs when you set a breakpoint and the debugger can't find it. The most common cause of an unresolved breakpoint is making a typing error when setting a symbolic breakpoint.

LLDB breakpoints have one or more locations. File line breakpoints have one location: the line where you set the breakpoint. Regular expression breakpoints are the ones most likely to have multiple locations because multiple functions can match the regular expression. Below the breakpoint is a listing for each location that shows the following information:

- A location number, such as 1.1.
- Where, which is usually the function name, file, and line number.
- The memory address of the location.
- Whether or not the location has been resolved by the debugger. An unresolved breakpoint location most likely will not be reached.
- Hit count, the number of times the breakpoint has been reached.

Use the `watchpoint list` command to examine your watchpoints. LLDB displays the following information for each watchpoint:

- The watchpoint number.
- The memory address of the watchpoint.
- The size, in bytes.
- State, which is enabled or disabled.
- Type, which can have the following values: `r` for read-only, `w` for write-only, and `rw` for read and write.
- The file and line number for watchpoints set on variables.

Disabling and Deleting Breakpoints

The `breakpoint disable` command disables a breakpoint. Supply a breakpoint number or a range of breakpoint numbers. If you supply no breakpoint numbers, LLDB disables all breakpoints.

```
breakpoint disable 2
breakpoint disable 5-9
breakpoint disable 4.2
```

The `breakpoint enable` command enables a disabled breakpoint. Supply a breakpoint number or a range of breakpoint numbers. If you supply no breakpoint numbers, LLDB enables all breakpoints. The following commands enable the breakpoints that were disabled in the previous example:

```
breakpoint enable 2
breakpoint enable 5-9
breakpoint enable 4.2
```

The `breakpoint delete` command deletes a breakpoint. Supply a breakpoint number or a range of breakpoints. If you supply no breakpoint numbers, LLDB deletes all breakpoints.

```
breakpoint delete 2
breakpoint delete 5-9
breakpoint delete 4.2
```

Use the `watchpoint disable`, `watchpoint enable`, and `watchpoint delete` commands to disable, enable, and delete watchpoints. Supply a watchpoint number or a range of watchpoint numbers. If you supply no watchpoint numbers, LLDB disables, enables, or deletes all watchpoints.

```
watchpoint disable 3-4
watchpoint enable 3-4
watchpoint delete 5
```

Breakpoint Commands

Breakpoint commands run a series of debugger commands when you reach a breakpoint. Run the `breakpoint command add` command to create a breakpoint command. There are two main ways to create a breakpoint command: using LLDB commands and using Python. Using LLDB commands is currently easier because information on using LLDB with Python is scarce. If you supply no flags to the `breakpoint command add` command, your command uses LLDB commands. Add the `-s` flag with the option `python` to build a breakpoint command using Python.

```
breakpoint command add BreakpointNumber (LLDB commands)
breakpoint command add -s python BreakpointNumber (Python)
```

When you run the `breakpoint command add` command, the console prompt switches to an interpreter. Enter the commands you want in the breakpoint command. If you're entering LLDB commands, enter one command per line. Enter `DONE` when you're finished. When you reach the breakpoint, LLDB will run the breakpoint command you created.

You can also create a command list in a text editor and load the file in the console to execute a breakpoint command. Enter one command per line in the text file. Run the `command source` command to run the commands in the command list. Supply the name of the file. The following command runs the command list saved in the file `MyCommands.txt` on the startup disk:

```
command source /MyCommands.txt
```

If you want to see a list of breakpoint commands for a given breakpoint, run the `breakpoint command list` command and supply a breakpoint number.

```
breakpoint command list BreakpointNumber
```

To remove a breakpoint command, run the `breakpoint command delete` command and supply a breakpoint number.

```
breakpoint command delete BreakpointNumber
```

Command Aliases

A command alias lets you create an abbreviation for a long command. Use the `command alias` command to create a command alias. The following example creates an alias where entering `?` invokes the `help` command:

```
command alias ? help
```

For more complicated aliases, you can enter placeholders for command arguments. The first argument is `%1`, the second argument is `%2`, and so on. The following command alias logs a user-defined number of items from memory to the console and gives the alias the name `memlog`:

```
command alias memlog memory read -c %1
```

You would log the contents of memory by running the `memlog` command and supplying the number of items and a starting memory address. The following example logs 64 items at memory location `0x12345678` to the console:

```
memlog 64 0x12345678
```

To remove an alias, run the `command unalias` command. Supply the name of the alias. The following example removes the `memlog` alias I created earlier:

```
command unalias memlog
```

Examining Variables

LLDB provides two commands to display variables. The first command is the `frame variable` command. If you supply no arguments, the `frame variable` command displays all local variables and function arguments. Supply a variable name to display that variable.

```
frame variable VariableName
```

Suppose you set a breakpoint inside an Objective-C class that contains a `NSTextView` object called `textView`. The following command prints the value of the `textView` member:

```
frame variable self->textView
```

The `frame variable` command has the following options:

- The `-D` option lets you limit the depth to display when showing a variable that consists of nested data structures. Supply a limit.
- The `-F` option displays the variables in a flat format.
- The `-G` option lets you specify a GDB format specifier string to format the variables' data.
- The `-L` option tells LLDB to display the variable's memory location.
- The `-O` option tells LLDB to display the variable as an Objective-C object.
- The `-P` option lets you supply the number of pointers to traverse when dumping the value of pointer variables. The `-P` option can help when displaying the contents of Objective-C objects, which normally consist of nested pointers.
- The `-T` option displays the variables' data types. LLDB normally shows the data types so you shouldn't need to explicitly set the `-T` option.
- The `-Y` option tells LLDB to omit summary information for the variables.
- The `-a` option tells LLDB not to show function arguments.
- The `-c` option tells LLDB to show the file and line number where the variable was declared.
- The `-d` option tells LLDB to show the object as its full dynamic type. Supply one of three values: `no-dynamic-values`, `run-target`, or `no-run-target`.
- The `-f` option lets you specify the format for displaying the output. Supply one of the viewing formats listed in Table 7.5. The `-R` option prints raw output with no formatting options.
- The `-g` option tells LLDB to display global and static variables.
- The `-l` option tells LLDB not to show local variables.
- The `-r` option lets you supply a regular expression to control the variables that are displayed.
- The `-s` option displays variable scope: local variable, function argument, global variable, or static variable.

The following command prints all local variables:

```
frame variable -a
```

The second command for displaying variables is the `print` command (shortcut `po`). Supply the name of a variable.

```
print variableName
```

When you run the `print` command, LLDB creates a convenience variable that starts with a dollar sign, such as `$1` or `$2`. Each time you print a variable, LLDB increments the number after the dollar sign. You can supply the convenience variable to the `print` command.

```
print $5
```

Examining Memory

Use the `memory read` command to examine memory contents. The `memory read` command has the following options:

- The `-c` option determines the number of items to display.
- The `-s` option lets you specify the size of an item in bytes. This option is the equivalent of the Byte Grouping setting in Xcode's memory browser.
- The `-l` option lets you specify the number of items that appear on one line in the listing.
- The `-f` option lets you specify the format to display the memory. Table 7.5 lists the formats. The `-R` option prints raw output with no formatting options.
- The `-o` option outputs the memory to a file instead of the console.
- The `-b` option saves the memory as a binary file.
- The `-A` option appends the memory to an existing file instead of overwriting the file.

The `memory read` command also has the `-D`, `-F`, `-G`, `-L`, `-O`, `-P`, `-T`, and `-Y` options. These options are identical to the corresponding options for the `frame variable` command.

The following example dumps 256 bytes of memory in 64 four-byte units starting at memory location `0xa0a73760` to a file called `MemoryDump.txt` on the startup disk:

```
memory read -c 64 -s 4 -o /MemoryDump.txt 0xa0a73760
```

Executing Shell Commands

Use the `platform shell` command to execute a Unix shell command in LLDB. The command takes the following form:

```
platform shell ShellCommand
```

With the shell command you can run another command-line program in Xcode's console. The following command runs the `leaks` tool on an application named `MyApp`:

```
platform shell leaks MyApp
```

Table 7.5 LLDB Memory Viewing Formats

Format	Description
b	Display as a binary (base 2) integer.
B	Display as a Boolean value.
c	Display as a character.
C	Display as a printable character.
d	Display as a signed decimal integer.
f	Display as a floating-point number.
o	Display as an octal (base 8) integer.
s	Display as a C string.
u	Display as an unsigned decimal integer.
x	Display as a hexadecimal (base 16) integer.
y	Display the memory as a series of bytes.
Y	Display the memory as a series of bytes with a listing of ASCII characters at the end of each display line. This is the default format and the format Xcode's memory browser uses.
E	Display as an enumeration.
O	Display as an OSType.
U	Display as a 16-bit Unicode character.
p	Display as a pointer.
a	Display as a character array.
A	Display as an address.
X	Display as a hex float.
i	Display as a machine instruction.
F	Display as a C++ complex float.
I	Display as a C++ complex integer.
intX_t[]	Display as an array of X-bit signed integers, where X can be 8, 16, 32, or 64.
uintX_t[]	Display as an array of X-bit unsigned integers, where X can be 8, 16, 32, 64, or 128.
floatX[]	Display as an array of X-bit floating-point numbers, where X can be 32 or 64.

LLDB Expressions

LLDB includes an expression parser, which allows you to interact with your code from inside LLDB. LLDB's expression parser uses Objective-C++, which means any Objective-C, C++, or C code is acceptable to place in an expression.

Run the `expression` (shortcut `expr`) command to add an expression. Enter your expressions in the console. Enter a blank line when you're finished. The following example adds an expression local variable `x` to your code:

```
expression
int x = 3;
x = x + 2;
```

To make the expression persist throughout the debugging session, add a `$` before the variable name to create a user variable. The following example creates a user variable `$x`:

```
expression
int $x = 3;
$x = $x + 2;
```

You can supply expression local variables and user variables to LLDB commands. The following command prints the value of the `$x` user variable I created in the previous example:

```
print $x
```

When debugging, the items of greatest interest are your application's variables and functions. LLDB's expression parser allows you to access in-scope variables, global variables with debug information, functions with debug information, and global symbols without debug information. Examples of in-scope variables are local variables and function arguments in the current function. Local variables in other functions are out of scope.

To access a global symbol without debug information, you must do a C type cast. When accessing a function without debug information, you must place the function's return type in parentheses. When accessing a variable without debug information, you must place its data type in parentheses.

```
(void)MyFunction;
(int)myIntVariable;
```

Logging

LLDB supports logging, but I recommend not using it initially. LLDB logs lots of internal state activity, such as when the debugger pauses and resumes a thread. Logging internal state activity creates a large amount of logging data to sift through, which is why I recommend not logging when you start debugging your application. Save the logging for nasty bugs where LLDB's other tools failed to find the cause of the bug.

Run the `log enable` command to turn on logging. The `log enable` command takes the following form:

```
log enable [Options] [Log Channel] [Log Category]
```

LLDB has the following log channels: `lldb`, `gdb-remote`, `kdp-remote`, and `dwarf`. Running the `log list` command shows the categories for each log channel. The `all` category logs everything. The `default` category uses the default set of logging categories. I recommend using the `-f` option to log output to a file. LLDB can log tens of megabytes of information, which will litter the console with log messages and can even bring up the spinning cursor. Logging to a file gives you a smoother debugging experience and lets you view the log data in a text editor.

The following command logs everything in the `lldb` channel to a file called `Log.txt` on the startup disk:

```
log enable -f /Log.txt lldb all
```

Use the `log disable` command to stop logging. Supply the log channel and log category you supplied when calling `log enable`. The following command disables the log I enabled in the previous example:

```
log disable lldb all
```

Chapter 8

Version Control

A version control system tracks the changes made to files and who made the changes. Version control comes in handy on medium to large projects and projects with multiple developers. Each developer can take an individual file, add code to the file, and the version control program records the code each developer added to the file. Even if you're working on a project by yourself, version control can help you. If you've ever mistakenly saved a file and wished you could go back to the way the file was before you saved it, you'll appreciate version control. With version control you can go back to an older version of a file.

Xcode 4 supports two version control systems: Subversion and git. The version control system you choose is a matter of personal preference, but if you're not sure, go with git. Xcode can create a git repository for you when you create a new project, which simplifies things greatly.

To start using version control in Xcode, you must perform the following tasks:

- Create a repository.
- Configure the repository so you can access it in Xcode.
- Import your Xcode project into the repository.
- If you have a Subversion repository, you must check out your project's files, which gives you a local copy of the project for you to modify.

Once you get version control working in Xcode, you can use Xcode to perform the following version control tasks:

- See what files have changed.
- Add files to a repository.
- Remove files from a repository.
- Compare two versions of a file.
- Commit changes to a repository.
- Discard changes you made to a file.
- See all the changes you've made to a file.
- See who modified a line of code.
- Add branches to your repository.

Creating a Repository

A *repository* is where a version control system stores the files you place under version control. You can create two types of repositories: local and remote. A local repository resides on your computer and will be used only by you. A remote repository allows other people to access the repository and check out the repository's files from their computers.

Creating a Local git Repository

The easiest way to create a local git repository is to create a project in Xcode. At the bottom of the new project's Save panel, you will see a checkbox named Create local git repository for this project. Select this checkbox, click the Create button, and you've created a git repository.

If you have an existing Xcode project you want to place under version control, launch the Terminal application. Navigate to the folder where your Xcode project resides and run the `git init` command.

```
git init
```

Creating a Local Subversion Repository

To create a local Subversion repository, launch the Terminal application and run the `svnadmin create` command. Supply a path to the repository. The following command creates a local repository named `MyRepository` on the startup disk:

```
svnadmin create /MyRepository
```

Creating a Remote git Repository

In git there is little difference between local and remote repositories. To create a remote repository, create a local repository and host it on a network or on the Internet. Other people clone your remote repository, which creates a local repository on their computers. They push their changes back to your remote repository.

Creating a Remote Subversion Repository

In Subversion creating a remote repository is the same as creating a local repository: run the `svnadmin create` command. To let other people access the repository, you must setup a Subversion server. You have two options for Subversion servers: use Subversion's `svnserve` server or use a Web server like Apache. A Web server is the better choice if you want other people to access your repository. Setting up a Subversion Web server is beyond the scope of this book, but there are three options to simplify the process for you.

Option 1: If you currently have a website, your hosting company has already set up a Web server for you. All you have to do is create a Subversion repository by running the `svnadmin create` command. Most web hosting companies have instructions for setting up Subversion on their websites.

Option 2: If you have an open source project, have it hosted on SourceForge or Google Code. They have tools to set up a Subversion repository for your project.

Option 3: If you want to set up a Subversion Web server on your Mac, use MAS. MAS is a Mac open source program that does everything you need to create a Subversion server. You can find a link to MAS on the *Xcode Tools Sensei* site under Resources.

How Many Repositories Should You Make?

You can create one repository per project or place multiple projects in one repository. Git works best with one repository for each project. Unless you have a group of related projects that will always be related, stick with one repository for each project if you're using git. For Subversion repositories, the number of projects in a repository is a matter of personal preference. If you're going to create branches in your Subversion repository, placing each project in its own repository makes things easier for you when working with the repository in Xcode.

Ignoring Files

When you place an Xcode project under version control, the version control system tracks every file and folder in the project. Obviously you want the version control system to track your source code files, but Xcode projects contain files and folders you may not want to track. How do you tell the version control system to ignore the files you don't care about versioning?

Create an ignore file, which is a text file that contains the files you want the version control system to ignore.

Naming the Ignore File

If you have a git repository, name the file `.gitignore`. If you have a Subversion repository, you can give it whatever name you want.

What Files Should Be Ignored?

What files and folders should be placed in an ignore file? First, look for any derived data or build folders in your project folder. Xcode 4 places its derived data outside of your project folder so you shouldn't have to worry about derived data if you created your project in Xcode 4. If you created your project in an older version of Xcode, it may have a build folder inside the project folder. This build folder should be in the ignore file.

A second category of files to ignore are the files inside Xcode project files. The Xcode project file is a file package. Select a project file from the Finder, right-click, and choose Show Package Contents to view the files in the package. You can ignore many of the files in the package. The `project.pbxproj` file is the only file in the project file package that should always be in version control. What you do with the other files in the project file package is a matter of personal preference. The following listing ignores files with the `.pbxuser` extension:

```
*.pbxuser
```

A third category of files to ignore are `.DS_Store` files. A `.DS_Store` file is a hidden Mac OS X file that stores a folder's custom attributes. The following listing ignores `.DS_Store` files:

```
.DS_Store
```

Add one line to the ignore file for each file type you want to ignore.

What to Do with the Ignore File?

If you have a git repository, place the `.gitignore` file in your repository. Git ignores the file types in the `.gitignore` file.

If you have a Subversion repository, launch the Terminal application and run the `svn propset` command on the `svn:ignore` property. Use the `-F` flag and supply the name of the ignore file. The following example uses an ignore file called `svnignore.txt`:

```
svn propset svn:ignore -F svnignore.txt .
```

Configuring the Repository for Xcode

After creating your repository, you must configure it so you can use it in Xcode. Open the Organizer by choosing Window > Organizer. Click the Repositories button in the Organizer toolbar. The repositories window will open in the Organizer. On the left side of the window is a list of repositories that Xcode has found. If the repository you want to use is not in the list, click the + button and choose Add Repository to add the repository.

When you choose Add Repository, a sheet opens. You must supply three pieces of information: the name of the repository, its location, and its type. Give the repository any name you wish.

The location depends on the type of repository you want to access. For a local git repository, supply the path to the repository.

```
/path/to/your/repository
```

A local Subversion repository has the `file://` prefix before the repository path.

```
file:///path/to/your/repository
```

A repository hosted on a Web server has either `http://` or `https://` as the start of the URL.

```
http://svn.example.com
```

A git repository hosted on github takes the following form:

```
git@github.com:githubRepositoryOwnerName/RepositoryName.git
```

A Subversion repository on a `svnserve` server has `svn://` as the start of the URL.

```
svn:///path/to/repository
```

To access a `svnserve` repository using `ssh`, enter `svn+ssh://` as the start of the URL.

```
svn+ssh:///path/to/repository
```

Choose the version control system you're using from the Type menu. Click the Add button to finish. Adding a remote Subversion repository has a second step, where Xcode asks you for paths to the Trunk, Branches, and Tags folders. If you don't know the paths, don't enter anything. You can add paths later by selecting the repository from the repository list.

Cloning Repositories

Sometimes you may want to work on a project located in a repository you can't directly access. An example is an open source project. On an open source project anyone can download the source code, but not everyone has direct access to the repository. The solution is to clone the repository, which creates a copy of the repository on your computer.

To clone a repository, click the + button in the lower left corner of the Organizer and choose Checkout or Clone repository. You will be asked for the location of the repository you want to clone. Click the Next button. Name the repository and choose a version control system from the Type menu. Click the Clone button. Pick a location to save the cloned repository. Click the Clone button to finish.

For some git repositories when you enter the location, the Next button's text changes to Clone. Click the Clone button. An Open panel opens. Name the repository, choose a location to save it, and click the Clone button.

If the repository you are cloning is a Subversion repository, the Clone buttons have the name Checkout. Checking out a repository is the Subversion equivalent of cloning a git repository. When you click the Checkout button, Xcode copies the files in the repository to your Mac.

Repositories Window

The repositories window is where you perform most actions on your repositories. You can do things like add repositories, examine commit messages, add branches, and see the files that are in the repository.

Click the Repositories button at the top of the Organizer to open Xcode's repositories window, which you can see in Figure 8.1. The repositories window has three sections: repository list, detail view, and history.

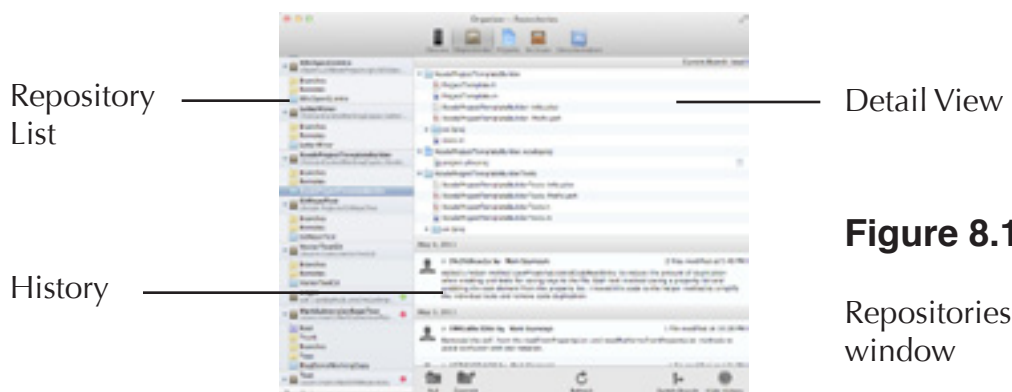


Figure 8.1

Repositories window

At the bottom of the repositories window are buttons to perform actions on repositories. The buttons that appear depend on the folder you select from the repository list and the version control system you're using. Some actions you can perform include adding branches, removing branches, switching branches, importing projects, checking files out of the repository, and adding directories to a repository.

Repository List

The repository list has one entry for each repository that shows the name of the repository and the path to the repository. There may be a colored dot next to the repository name. The color of the dot represents the repository's status. Green means the repository is online. Red means Xcode can't find the repository, which can happen if the repository doesn't exist, if there is a mistake in the path to the repository, or if you moved the repository. Yellow (it may look yellow orange) means the repository has an authentication problem. A Subversion repository may have yellow status if you don't set the paths for the branches, tags, and trunk folders. Sometimes local git repositories have a yellow status, but you can ignore the status because it is a cosmetic issue. You can still access the repository.

Each repository has a disclosure triangle next to it. Clicking the triangle shows the repository's folders. A git repository has three folders: a Branches folder, a Remotes folder, and a working copy folder that usually has the same name as the project. If you have a Subversion repository, there will be no Remotes folder. A Subversion repository may have additional folders, depending on how you set up the repository. Some common folders in a Subversion repository are Root, Trunk, and Tags.

Detail View

What the detail view shows depends on the folder you select. If you select the repository, the detail view shows information about the repository. For a Subversion repository, you can set a user name and password as well as the location to the Trunk, Branches, and Tags folders. If you select the Branches folder, the detail view lists the repository's branches. Selecting the Branches folder lets you add branches to a repository, remove branches from a repository, and checkout a branch from a Subversion repository.

Selecting the Remotes folder shows any remote branches the git repository has. You can add and remove remote branches as well as track them. The "Tracking Branches" section later in this chapter has more information on tracking remote branches.

If you select a working copy folder, the detail view shows the files in the folder along with their version control status. These files are the files that are in the repository, the files that are under version control.

History

The history shows the commits you made to the repository. How much history is shown depends on what you select from the repository window and detail view. If you select the repository itself, the whole repository's history is available. If you select a branch from the detail view, the branch's history appears. If you select an individual file, that file's history appears.

For each commit the history tells you the version number, the person who made the commit, the commit message, the number of files modified, and when the commit occurred. Each commit has a disclosure triangle next to it. Clicking the triangle shows the files that changed along with a View Changes button. Clicking the button opens a sheet that shows what changed during that commit.

Each commit displays the committer's avatar on the left side. Clicking the avatar opens a pop-up editor that lets you modify the address book entry for the person who made the commit.

Selecting a person's name in a history entry makes a pop-up button appear. Click the button to send an email to that person.

Importing Your Project to the Repository

After creating your repository you must get your Xcode project and its files into the repository. If you told Xcode to create a local git repository when you created your project, you don't have to worry about importing your project to the repository. Xcode imported it for you.

Importing to a git Repository

The fastest way to import your project to a git repository is to launch the Terminal application. Navigate to the project folder and run the `git add` command. The following command adds all the files in the project folder to a git repository:

```
git add .
```

Alternatively, you can add files to a git repository from Xcode. Select a file from the project navigator, right-click, and choose Source Control > Add. To finish adding the file, commit the file by right-clicking and choosing Source Control > Commit.

Importing to a Subversion Repository

Importing a project to a Subversion repository is a little more complicated. If you follow Subversion conventions, go to your project folder in the Finder and add three folders named branches, tags, and trunk. Adding the branches, tags, and trunk folders is not a mandatory step to add a project to a Subversion repository, but doing so will make your life easier if you need to branch your code in the future. Suppose you have a Mac OS X application, and you decide to create an iOS version. You could use the same Subversion repository, but have two branches: one for the Mac version of your code and one for the iOS version. I recommend adding the branches, tags, and trunk folders just to be safe.

If you decide to add the branches, tags, and trunk folders to your project folder, you should move the files in the project folder to the trunk folder. If there is a build folder inside your project folder, you should move the build folder out of the project folder when adding the project to the repository. The build folder contains files Xcode creates when it builds your project, such as object and executable files. The contents of the build folder can become large as you work on your project, and the build folder's contents are files you don't want to place under version control because the files' contents are going to change every time you build your project. A project named MyProject should have the following directory structure prior to import:

```
> MyProject
  > branches
  > tags
  > trunk
    > Everything in your project except the build
      folder. Move the build folder out of
      MyProject.
```

To import a project, select the Root folder for the repository on the left side of the repositories window. Click the Import button at the bottom of the repositories window. An Open panel opens. Select your project and click the Import button. A dialog opens for you to enter a commit message. Click the Import button in the dialog to finish importing the project.

After clicking the Import button, there should be a directory inside the Root folder with the name of your project. Inside the project name directory should be the branches, tags, and trunk folders. The trunk folder should contain your project's files. If these folders don't show up, click the Refresh button to refresh the repositories window.

Checking Out Files from a Subversion Repository

After importing a project to a Subversion repository, check out the files in your project so you can use them in Xcode. Select the Root folder from the left side of the repositories window. Select your project's trunk folder from the detail view, and click the Checkout button at the bottom of the repositories window.

A Save panel opens asking you where you want to store the checked out files on your hard drive. Navigate to where you want to store the checked out files. I recommend creating a folder specifically for storing the working copies of your Xcode projects. Click the Checkout button.

Remember where you checked out the files after performing the checkout. The folder where you checked out the files contains the files that are under version control. The files that are under version control are the files you should use in Xcode. Why is it so important to remember where you checked out the files?

Remembering where you checked out the files is important because after the checkout, you have two copies of your Xcode project and source code files: the one you initially created (Copy A) and the one you checked out (Copy B). Copy B is the one that is under version control. Copy B's project file is the one you should open in Xcode. If you open Copy A's project file, you won't be able to do any version control operations because that copy isn't under version control. You should move Copy A to a backup disk, then delete Copy A to avoid confusion.

Seeing Which Files Have Changed in Your Project

Initially the files in your project match the files in the repository. As you work on your project, editing source files and adding files to the project, some files no longer match. These files are the ones you must update in the repository. To see the list of files that don't match the saved version in the repository, look at the project navigator. To the right of each file is a one-character code representing the file's version control status. Table 8.1 contains a list of status codes.

If you modified some files in your project, but the project navigator shows every file to be up-to-date, make sure the project is under version control. Open the Organizer and click the Repositories button in the Organizer toolbar. Make sure the repository for your project is on the list at the left and there's no red or yellow dot next to the repository.

Table 8.1 Version Control Status Codes

Code	Description
?	Unversioned, which means the file is not in the repository. You must add the file to the repository from Xcode.
-	The file is in a folder that is not in the repository. You must add the folder to the repository from Xcode's repositories window or from the command line.
M	You modified the file. Commit the file to add the changes to the repository.
A	To be added. The file will be added to the repository the next time you commit the file to the repository.
D	To be deleted. The file will be removed from the repository the next time you commit the file to the repository.
C	The changes you made to the file may conflict with changes made in the latest version. You would get this code if another programmer modified a file while you had that file checked out.
U	The version of the file you're using is older than the latest version in the repository.
Blank	The file matches the latest version in the repository.

The file inspector also shows a file's source control status. Choose View > Utilities > Show File Inspector to show the file inspector. The Source Control section in the file inspector shows you a file's version control status, location on disk, and the latest version number of the file in the repository.

Adding Files to the Repository

To add files to your repository, add the files to the project. These files have the status A in the project navigator, which means they're ready to be added to the repository. Select the recently added files, right-click, and choose Source Control > Commit Selected Files to add the files to the repository.

If the files have the status unversioned (it has a question mark) in the project navigator, select the files, right-click, and choose Source Control > Add. Choosing Source Control > Add changes the status from unversioned to ready to add.

Removing Files from the Repository

Although it's more common to add files to a project than to remove them, you can remove files from a repository from Xcode. To remove files from a git repository, perform the following steps:

1. Select the files you want to remove from the project navigator.
2. Press the Delete key.
3. An alert opens asking if you want to move the deleted files to the trash or remove references to the files from the project. Click the Move to Trash button. If your version of Xcode has a Delete button instead of a Move to Trash button, click the Remove Reference button.
4. If you clicked the Remove Reference button, move the files you want to delete to the Trash. You won't be able to remove the files from the repository from Xcode unless you move them to the Trash.
5. Choose File > Source Code > Commit to remove the files from the repository.

To remove a file from a Subversion repository, perform the following steps:

1. Open the Organizer and click the Repositories button.
2. Select the repository's Trunk folder from the repository list.
3. Select the file you want to delete in the detail view.
4. Click the Delete button at the bottom of the Organizer to remove the file from the repository.

Seeing the Changes You Made to a File

A common source code management task is comparing two revisions of a file. When you commit a change to the repository, you want to see what changed so you can write an informative message when you commit the change.

To see the changes you made to a file, select the file from the project navigator and open the version editor. On the right side of the project window toolbar is the Editor group of buttons. Click the right button to open the version editor, which you can see in Figure 8.2.

The version editor shows two versions of the file. Initially the left pane shows the local version of the file, and the right pane shows the base version, which is the latest version of the file in the repository. If you want to compare the local version to an earlier version of the file, click the jump bar under the right pane. Clicking the jump bar opens a menu that gives you access to every version of the file in the repository. Choose the version you want. You can also use the timeline button to choose a file version. The advantage of using the timeline button is it shows the commit messages for each version. In the timeline view, the oldest versions are at the top of the timeline.

Xcode highlights the file's changes in the version editor. If the file is large enough to require a scroll bar, the scroll bar has a red line where there's a change.

Committing Changes You Made

You've made changes to one of your source code files. It could be new code, a bug fix, a speed boost, or taking some ugly code that somehow works and cleaning it up. You tested the changes you made, and everything works perfectly. At this point you should commit your changes, which creates a new version of the file in the repository.

The easiest way to commit the changes to a single file is to select the file in the project navigator, right-click, and choose Source Control > Commit Selected Files. A sheet opens where you can see what changed since the last version and enter a commit message that explains the changes you made. Click the Commit button to add a new version of the file to the repository.

If you have a bunch of files you want to commit at once, choose File > Source Control > Commit. A sheet opens with a list of files in your project that aren't up-to-date. To the left of each file is a checkbox. Selecting the checkbox tells Xcode to commit the file. To the right of each file is its source control status. Enter a commit message and click the Commit button.

Xcode 4.4 adds the ability to cherry pick commits. Cherry picking allows you to commit some changes to a file while leaving other changes uncommitted. If you make five changes to a file, but only want to commit three, cherry pick the commits.

To cherry pick commits start by committing the file. A sheet opens that shows the changes. Between the two versions of the file is a button with a number. There is a button for each change in the file. For each change you don't want to commit, click the button and choose Don't Commit. Click the Commit button to commit the changes.

Timeline View

Timeline Button



Figure 8.2

Version editor

Discarding Changes

Suppose you made some changes in a file and found out the changes didn't work. How do you go back? Select the file in the project navigator, right-click, and choose Source Control > Discard Changes. An alert opens asking if you're sure you want to discard the changes. Click the Discard Changes button to finish the discard.

When you discard changes Xcode erases all the changes you made and takes you back to the last version you sent to the repository. Discarding changes is the solution when you save a file and wish you hadn't.

Xcode 4.4 adds the ability to discard changes to a file from the version editor. The version editor has a button for each change in the file. The button is between the two versions of the file and has a number. You won't be able to see the buttons if you're showing the timeline. Click the timeline button to hide the timeline. Click the button and choose Discard Change to discard the change. Using the version editor allows you to discard a single change to a file.

Viewing Annotations

When multiple people make multiple changes to a file, it's nice to know who changed what in the file. Annotations perform this vital task, telling you the following information for each line in a source code file:

- The revision of the file that last modified the line of code.
- Who modified the line.
- When that person modified the line.

To view a file's annotations, select the file from the project navigator. Open the version editor and click the Blame button at the bottom of the version editor. The button is called the Blame button because annotations can be used to blame someone for creating a bug. Figure 8.3 shows an example of a file's annotations. The annotations appear to the right of the file. For each block of code you can see who edited the block and when the block was changed. If the code block is large enough, Xcode displays the commit message. Next to the person who edited the code block is a colored bar. The darker the color, the more recently the block was changed.

If you move the mouse cursor over an annotation, a small button appears next to the revision date. Clicking the button opens a pop-up editor that lets you read the commit message. The revision number appears below the person who modified the code. Next to the revision number is a focus button with an arrow. Clicking the button opens that revision in the version editor.

Viewing a File's Revisions

If you want to see all the revisions for an individual file, use the SCM log. To open a file's SCM log, select the file from the project navigator. Open the version editor and click the Log button at the bottom of the version editor. When you open the SCM log, shown in Figure 8.4, it lists every revision made to the file to the right of the file, with the most recent version at the top. The inspector shows the following information:

- The date.
- Who checked the revision into the repository. Click the person's name to open a menu that lets you email the person who checked in the revision.
- The time the person checked the revision into the repository.
- A message describing the changes made in this revision.

If you move the mouse cursor over a log entry, a small Info button appears next to the commit time. Clicking the button opens a pop-up editor that shows the revision number and the files that were modified in the revision. Next to the revision number is a focus button with an arrow. Clicking the button opens that revision in the version editor.

Branching

When you create a branch, you create a copy of your code so you can work on the code without messing up the stable code you have in the trunk (the master branch in git) of your repository. Branching is especially useful for experimentation. Create a branch and experiment. If the experiment works, merge the changes. If the experiment doesn't work, delete the branch. Even if you're not coding anything risky, branching can help. Many developers create a local branch to do their work and periodically merge their changes back to the trunk.

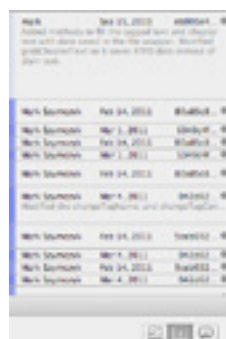


Figure 8.3

Annotations

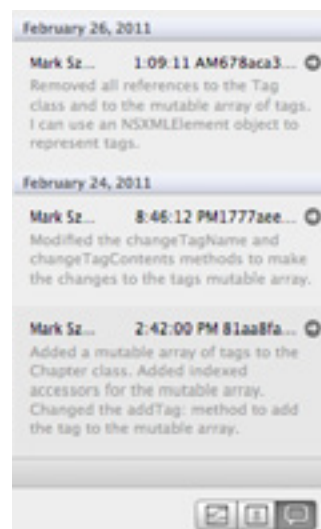


Figure 8.4

SCM log

Creating a Branch

Creating a branch is pretty simple in Xcode. Select the Branches folder for the repository and click the Add Branch button at the bottom of the Organizer. Name the branch and give it a starting point. At this point what you do depends on the version control system you're using. For a git repository, the starting point initially is master. Every git repository has a master branch. If you select the Automatically switch to this branch checkbox, Xcode makes the newly created branch the current branch. Click the Create button to finish creating the branch.

Xcode 4.4 adds the ability to create a new branch when committing to a git repository. When you commit a file, there is a Commit to Branch button in the lower left corner of the sheet. If you click the Commit to Branch button, a second sheet opens. Click the pop-up cell in the Branch column and choose New Branch to create a branch.

For a Subversion repository, the starting point initially is the trunk of the repository. To work with a local branch of a Subversion repository, you must checkout the branch. Select the Automatically checkout this branch checkbox and click the Create button. A Save panel opens where you specify a location to checkout the branch.

If you have a Subversion repository, there may not be a Branches folder underneath the repository on the left side of the Organizer. Select the repository and enter the relative path to your branches folder in the Branches text field. Make sure there is a green dot next to the text field after entering the path.

Removing a Branch

Removing a branch is even simpler than adding one. Select the Branches folder for your repository. Select the branch and click the Remove Branch button at the bottom of the Organizer.

Switching Branches

If you have multiple branches, you sometimes need to switch branches. To merge changes from a local branch to a git repository's master branch, you must switch from the local branch to the master branch. Xcode lets you switch branches from the Organizer.

Each repository listed in the Organizer has folders inside it. Click the disclosure triangle if you don't see any folders. You should see a working copy folder with the name of your project. Select this folder. Click the Switch Branch button at the bottom of the Organizer to switch branches. Choose a branch and click the OK button.

Merging

Choose File > Source Control > Merge to merge your changes to a particular branch. Before you do the merge, switch to the branch where you want to merge the changes. The merge destination for a git repository is usually the master branch. The merge destination for a Subversion repository is usually the trunk of the repository.

A sheet opens. If your project has multiple working copies, the sheet has a list of working copies. Select the checkbox next to your project's working copy. Choose a branch from the Branch menu and click the Choose button. Xcode will not merge a project if any files in the project have uncommitted changes. Commit the changes before merging.

Merges become more complicated when other people modify a file and their changes conflict with yours. When a merge has conflicts, a sheet opens after you specify a merge destination and click the Choose button. The sheet shows both versions of the file and highlights each conflict. Xcode displays a switch for each conflict. Use the switches to resolve the conflicts. Click the Merge button when you're finished resolving the merge conflicts.

Tracking Branches

A *tracking branch* is a branch in your local repository that tracks a branch in a remote repository. Tracking branches make working with remote git repositories easier in Xcode. You must be using git to use tracking branches in Xcode.

Creating a tracking branch is a two-step process. First, add a remote, which is the branch in the remote repository you are going to track. Second, track the remote you added.

If you look at the repository list on the left side of the Organizer, you should see a Remotes folder for your git repository. Select the Remotes folder. Click the Add Remote button at the bottom of the Organizer to add a remote. Enter a name and location for the remote. The location is the URL of the remote repository. Click the Create button to finish, which adds a branch to the list of remote branches.

To track a remote branch, select it and click the Track Branch button. Tracking a remote branch is similar to adding a branch to a local repository. A sheet opens. Name the branch and pick a starting location. You have the option to switch to the branch. Click the Create button to finish. If you select the Branches folder for your repository, you should see the tracking branch in the list of branches.

Pushing and Pulling

Pushing and pulling are operations you can perform on remote git repositories. Pushing moves your local changes to a remote repository. Pulling brings your repository up-to-date with a remote repository. When you pull, git fetches the contents of the remote repository and merges the changes with your local repository. Pushing and pulling is much easier in Xcode if you track a branch in the remote repository. Read the previous section to learn more about tracking branches.

To push to a remote repository, choose File > Source Control > Push. A sheet opens. If your project has multiple working copies, the sheet has a list of working copies to push. Select the checkbox next to your working copy, which is usually the name of your project. Choose a branch from the Remote menu and click the Push button. If the name of your tracking branch does not match the name of a branch in the remote repository, Xcode can create a branch in the remote repository with the name of your tracking branch. This branch appears in the Remote menu with (Create) after it.

To pull from a remote repository, choose File > Source Control > Pull. A sheet opens. If your project has multiple working copies, the sheet has a list of working copies. Select the checkbox next to your working copy. Choose a branch from the Remote menu and click the Choose button.

Your project must have no uncommitted changes to push or pull. Commit your changes before pushing or pulling.

Snapshots

If version control is overkill for your needs, take a look at Xcode's project snapshots. When you take a snapshot of your project, Xcode saves a copy of all the files in the project. If you make a bunch of changes to your code that don't work and you want to discard those changes, you can revert back to the snapshot you took.

Snapshots have two weaknesses that make them unsuitable as a replacement for version control. The first weakness is they can only be used by one person on their local Mac. If you have a team of developers, they can't use snapshots the way they would use version control. The second weakness is when you take a snapshot, Xcode saves a copy of all the files in the project, even if there were no changes in the files from the previous snapshot. If you take 25 snapshots of a project, there will be 25 copies of each file, which can take up a lot of space. Snapshots are good for solo developers working on small projects and for people to do quick experiments with code. Take a snapshot, experiment, and revert back if the experiment didn't work.

Taking a Snapshot

Taking a snapshot of a project is easy. Open the project and choose File > Create Snapshot.

Looking at a Project's Snapshots

To examine your project's snapshots, open the Organizer and click the Projects button in the Organizer toolbar. On the left side of the window is a list of projects. Select a project to see its snapshots, which you can see in Figure 8.5.

The bottom half of the window contains a list of snapshots. Double-clicking a snapshot opens a sheet that lets you see what files changed in this snapshot. Select a file to view its changes. Click the Cancel button when you're finished. Double-clicking the first snapshot you took of a project exports the snapshot. Choose a location to save the snapshot on your hard disk.

Restoring a Snapshot

To restore a snapshot for a project, choose File > Restore Snapshot. A sheet opens with a list of snapshots you took for the project. Select a snapshot to restore and click the Restore button. A larger sheet opens, showing the files that were modified in that particular snapshot. Select a file to view its changes. Click the Restore button to finish restoring the snapshot.

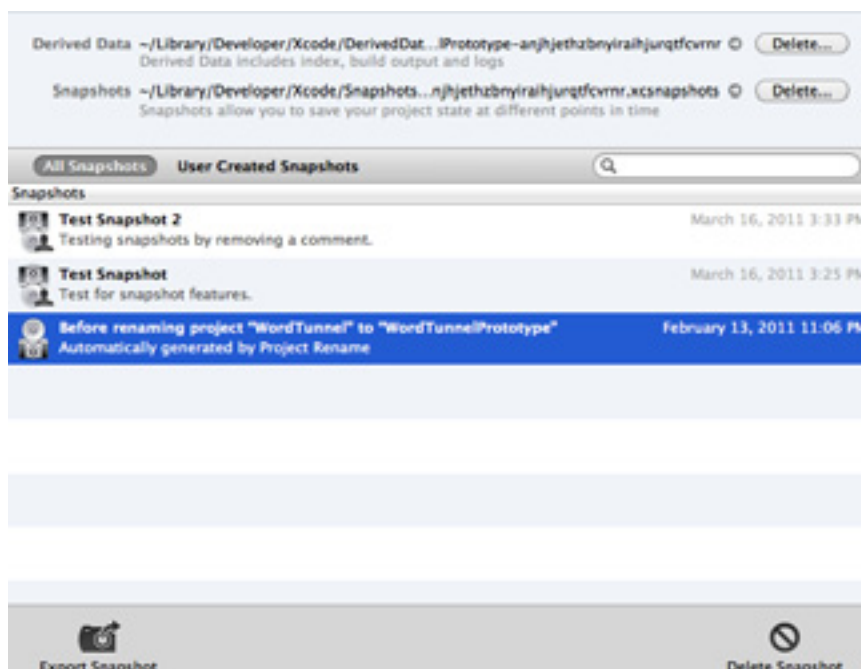


Figure 8.5

Snapshots window

If you want to restore a snapshot to a different location than the current project location, select the snapshot from the snapshot list in the Organizer and click the Export Snapshot button. A sheet opens, showing the files that were modified in that particular snapshot. Select a file to view its changes. Click the Export button, and you will be asked for a location to store the restored project.

Deleting Snapshots

To delete a snapshot, select the snapshot and click the Delete Snapshot button in the lower right corner of the Organizer. To delete all snapshots for a project, click the Delete button above the snapshot list.

Accessing Your Snapshots

You can find your snapshots in the following directory:

```
/Users/YourUsername/Library/Developer/Xcode/Snapshots
```

Above the project's snapshot list in the Organizer is the location of the project's snapshots and an inline button with an arrow. Clicking the button takes you to the snapshot location in the Finder.

If you want to store your snapshots in a different location, open Xcode's Locations preferences and choose Custom from the Snapshots pop-up menu. Enter the path to the location where you want Xcode to store the snapshots.

Chapter 9

Instruments

Instruments is a tool that dynamically traces your program's execution. You can use it for many purposes, including the following:

- Detecting memory leaks.
- Examining memory allocations.
- Profiling your application to see where it spends its time.
- Measuring how well your application takes advantage of multi-core processors.
- Tracing file activity.
- Recording keyboard and mouse events.

A nice feature of Instruments is you don't have to recompile your code to use it. As long as your program generates debugging symbols, you should be able to trace your program with no problems in Instruments.

Instruments comes with over 30 instruments. If none of the built-in instruments meets your needs, you can create your own.

Tracing From Xcode

The easiest way to get started with Instruments is to launch it from Xcode by choosing Product > Profile. Instruments launches. You will be asked to pick a template. Pick a template, click the Profile button, and you're tracing your Xcode project.

If you prefer not having to pick a template, modify your project's scheme to automatically profile with a certain instrument when you choose Product > Profile.

1. Choose Edit Scheme from the Scheme menu in the project window toolbar.
2. Select Profile from the left side of the scheme editor.
3. Choose an instrument from the Instrument pop-up menu.

When you choose Product > Profile, Instruments traces your project with the instrument you chose from the Instrument pop-up menu.

Creating and Setting up a Trace Document

To trace an application, you must create a trace document. After creating the trace document, you can add instruments to the trace, remove instruments from the trace, customize the graph, and open the detail view to configure the trace.

If you trace from Xcode, Instruments starts recording immediately, which means you don't get a chance to configure the trace before recording. Click the Stop button if you want to configure the trace.

Creating a Trace Document

When you launch Instruments, you will be prompted to select a template for a trace. Choosing File > New will also prompt you to select a template. There are four categories of templates: Mac OS X, iOS, iOS Simulator, and User. User templates contain templates that you create.

Select the template you want to use and click the Choose button to create the trace document. Don't fear picking the wrong template. You can add and remove instruments from the trace document after choosing a template.

If you want to start recording right away, click the Record button after choosing a template. When you click the Record button, one of two things will happen. A sheet will open for you to choose an application to trace or Instruments will start recording all running processes. The behavior you get depends on the template you choose, but most templates ask for an application to trace when you click the Record button.

Mac OS X Templates

Instruments comes with the following templates for Mac applications:

- Blank. Pick this template if you don't like any of the other templates.
- Allocations, which measures your application's memory usage and allows you to examine individual memory allocations.
- Leaks, which searches for memory leaks.
- Zombies, which uses the same instrument as the Allocations template, but it's also set to detect the release of `NSZombie` objects. The Zombies template alerts you when you access a freed object in your application and helps you locate where you are accessing the freed object in your source code.
- GC Monitor, which tracks garbage collection in Objective-C 2.0 programs.
- Activity Monitor, which traces overall system activity, including CPU, memory, and disk usage.
- Time Profiler, which measures where your application spends its execution time.

- Dispatch, which monitors dispatch queue activity. It works with Apple's Grand Central Dispatch technology, which makes it easier for applications to utilize the power of multi-core processors.
- Multicore, which measures how well your application takes advantage of multi-core processors.
- System Trace, which records thread scheduling behavior, system calls, and virtual memory operations.
- File Activity, which traces file activity that is not related to Core Data, such as opening, closing, reading, and writing files.
- Core Data, which traces Core Data file activity.
- UI Recorder, which records all user events, such as key presses and mouse clicks.
- Sudden Termination, which tests how your application handles sudden termination. Sudden termination was added in Mac OS X 10.6 to make system shutdown faster.
- Cocoa Layout, which helps you debug problems with Cocoa's auto layout. You must be running Mac OS X 10.7 or later to have this template.
- Event Profiler, which records a sample when a low-level event occurs, such as a L2 cache miss. You specify the event to sample.
- Counters, which collects performance monitor counter (PMC) events from the CPU. You specify the events to collect.

iOS Templates

Instruments has Blank, Activity Monitor, Time Profiler, Leaks, System Trace, and Allocations templates customized for iOS applications as well as the following templates:

- Automation, which runs a script that simulates user interaction in your iOS application.
- Energy Diagnostics, which measures energy usage.
- System Usage, which records I/O system activity.
- Core Animation, which measures graphics performance.
- OpenGL ES Driver, which measures graphics performance for iOS applications that use OpenGL ES.
- OpenGL ES Analysis, which measures OpenGL ES performance and provides recommendations to fix any performance problems in your application.
- Network, which measures how your application uses network connections.

To use the iOS templates I listed, your application must be running on a device and the device must be connected to your Mac. The Record button is disabled if you use these instruments on an application running in the iOS Simulator.

If you want to measure your application's OpenGL ES performance, use the OpenGL ES Analysis template. The OpenGL ES Analysis template includes the OpenGL ES Driver instrument. The only reason to use the OpenGL ES Driver template is if you want to profile your application as well as measure OpenGL ES performance. The OpenGL ES Driver template includes the Time Profiler instrument.

iOS Simulator Templates

Instruments provides the following templates for the iOS Simulator:

- Blank
- Allocations
- Leaks
- Activity Monitor
- Zombies
- Time Profiler
- Automation
- File Activity
- System Trace
- Core Data

The iOS Simulator templates work similarly to their Mac OS X counterparts, with the exception of the Automation template, which works similarly to its iOS counterpart.

Trace Document Window

When you pick a template, Instruments opens a trace document window for you. Figure 9.1 shows the trace document window after a trace. Your trace document window looks slightly different because you haven't traced anything yet. The trace document window has the following sections:

- Toolbar
- Instrument list
- Track pane
- Detail view
- Extended detail view

The toolbar has controls to record, pause, and stop tracing. It also lets you pick the programs to trace and lets you show and hide the detail view and extended detail view. The instrument list shows all the instruments being used in the trace.

The track pane and detail view both display trace statistics. The difference is the track pane graphs the statistics over time while the detail view shows the statistics in a table.

The extended detail view shows additional information. For most instruments, the extended detail view shows the call stack.

Adding and Removing Instruments

After selecting a template, you can add instruments to the trace document as well as remove them from the trace document. Click the Library button in the trace document window toolbar to open the Library, which contains a list of available instruments. To add an instrument to the trace, drag an instrument from the Library to the trace document window. To remove an unwanted instrument from the trace, select the instrument from the instrument list and press the Delete key.

Selecting an instrument from the Library makes a description appear at the bottom of the window. The description tells you whether the instrument is for Mac applications, iOS applications, or both. iOS developers should choose iOS from the pop-up menu at the top of the Library to see all the available instruments for iOS applications.

Once you set the instruments you can make your trace a template by choosing File > Save As Template. You will be asked for a name and a description. When you create a new trace, your template appears in the User section.

Customizing the Track Pane

Each instrument in the trace has an Info button next to it. Clicking the button opens a pop-up editor that lets you customize what the track pane displays after running the trace. You can choose the graph style, graph type, zoom level, and the statistics that appear in the graph. Not every instrument has statistics to configure. The zoom level in the pop-up editor zooms the track pane vertically. The slider below the instrument list zooms the track pane horizontally.

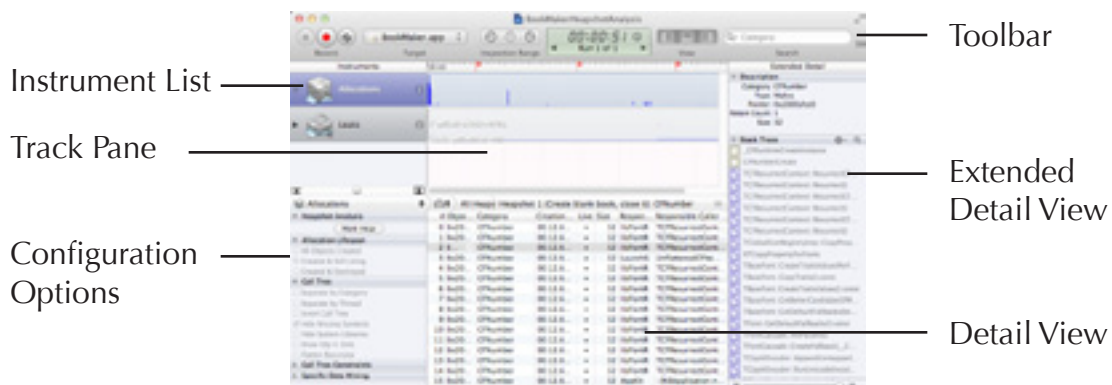


Figure 9.1

Trace document window

Some instruments have configuration options as well as graph customization options. The Time Profiler instrument lets you specify the sampling rate. The Allocations instrument lets you customize the memory allocations it records. If an instrument has too many configuration options to fit in the pop-up editor, there will be a Configure button. Clicking the Configure button shows the extra options.

Showing the Detail View

Some instruments have configuration options in the detail view that must be set before tracing if you want to use them. Suppose you're checking for memory leaks. The Leaks instrument can be configured to periodically check for leaks and to gather the contents of any leaked memory. But you must configure the Leaks instrument before you start tracing for the configuration options to take effect.

If you don't see the detail view, the jump bar will be at the bottom of the trace document window. Click the jump bar or choose View > Detail to show the detail view. What you can configure in the detail view before tracing depends on the instrument.

Running a Trace

After setting up your trace, it's time to run it. But running your trace is more complicated than hitting the Record button. You must tell Instruments what to trace, and if you're tracing a single program, you must choose the program to trace.

If you trace from Xcode, Instruments starts recording immediately, which means you don't have to determine what to trace or choose a program to trace. Instruments traces your Xcode project.

Determining What to Trace

To run a trace the first thing you must do is tell Instruments what to trace. Use the Target pop-up menu in the trace document window toolbar to tell Instruments what to trace. You can trace all running programs, trace a program that is currently running, launch a program to trace, or use an instrument-specific target.

Not all instruments have all the tracing options. If you're checking for memory leaks, you can check on only one program so the option to trace all programs will be disabled. The Activity Monitor instrument traces all running programs, even if you tell it to trace one program.

Instrument-specific targets require more explanation. They let each instrument in the trace work with its own target. You can run one instrument on one of your applications and run a second instrument on another application in one trace. You can also trace all running programs in one instrument and trace your application in a second instrument. To set a target for an instrument, perform the following steps:

1. Choose Instrument Specific from the Target pop-up menu in the trace document window toolbar.
2. Select an instrument from the instrument list.
3. Click the Info button next to the instrument.
4. Use the Target menu in the selected instrument's pop-up editor to set the target for that instrument.
5. Repeat Steps 2-4 for the other instruments in your trace.

Choosing a Program to Trace

Most of the time you use Instruments, you're interested in tracing one application: the one you're currently developing. If your application is currently running, choose Attach to Process from the Target pop-up menu and pick your application.

If your application is not running, choose Choose Target from the Target pop-up menu. A submenu with a list of recently traced programs opens. Those of you running Instruments for the first time will find this menu empty. If the application you want to trace is not in the recently-traced programs submenu, choose Choose Target > Choose Target. Yes, there are two Choose Target menus.

When you choose Choose Target > Choose Target, a sheet opens for you to pick a program to trace. From this sheet you can also set environment variables, command-line arguments, and a working directory if you need them to run your application. Selecting the View All checkbox makes hidden files visible. Selecting the Traverse Packages checkbox lets you navigate the contents of bundles, which normally appear as a single file. After picking your application, click the Choose button.

Tracing

After choosing a program to trace, click the Record button to start tracing. Some instruments require you to enter your user password to run the trace. Run the program you're tracing. Click the Pause button to pause and resume recording. If you want to run multiple traces, click the Stop button (the Record button becomes the Stop button when you're recording) and click the Record button to start a new trace.

If you run multiple traces, remember that Instruments is initially set to save only the most recent run to disk when you save a trace. Deselect the Save Current Run Only checkbox in the Save panel to save all runs to disk.

Recording Options

Choosing File > Record Options allows you to set additional recording options. When you choose File > Record Options a sheet opens that lets you delay recording, set a time limit for recording, and defer the display of trace data. When you select the Deferred Mode checkbox, Instruments doesn't display any trace data until you stop recording. Deferring the display of trace data improves tracing performance, which makes your application more responsive during tracing.

Alternate Trace Document Views

Instruments has two alternate views of the trace document: a mini instruments window and a fullscreen window. The mini instruments window gets out of your way so you can see more of your application during tracing. The fullscreen window gives you the largest possible view of your trace.

Choosing View > Mini Instruments hides any open trace document windows and opens a mini instruments window. The mini instruments window has a list of all open trace documents and lets you start and stop recording. Click the Close button in the mini instruments window to return to the normal trace document window.

Choosing View > Full Screen makes the trace document window fill the screen. Those of you running Mac OS X 10.7 or later can click the fullscreen button in the upper right corner of the trace document window to go fullscreen. Fullscreen mode is more useful for viewing trace results than recording. Move the mouse cursor over the top of the screen to make the menu bar appear. Choose View > Full Screen a second time to exit fullscreen mode.

Examining Trace Results

After running your trace the next step is to examine the trace results. Instruments displays a lot of trace data and provides many options to filter the statistics it displays. The material in this section is one of the more difficult sections in this book; it was the most difficult section for me to write. You may need to read the material multiple times to fully understand it. I'm going to start with general information that applies to most instruments. After that I'll move on to explaining the results of specific instruments.

Track Pane

The track pane shows a graph of trace data statistics for each instrument in the trace. The statistics Instruments graphs depend on the instrument and what you told the instrument to graph when you configured the instrument. In most cases the track pane graphs activity in your application, such as memory and CPU usage.

The graph shows spikes in activity, but it doesn't tell you the amount of activity. The track pane's inspection flags tell you the amount of activity. Inspection flags appear when you click the timeline at the top of the graph and hold the mouse button down. The inspection flags are similar to tool tips. By dragging the mouse, you can view changes in the graphed statistics over time. If you're using the Leaks instrument, the inspection flags show you the number of leaks and the amount of leaked memory.

Choosing View > Increase Deck Size zooms the graph vertically. The slider below the instrument list controls the horizontal zoom. Move the slider to the right to zoom horizontally. Choosing View > Snap Track To Fit sizes the graph to fit in the trace document window.

Detail View

The detail view shows the trace data. The trace data can appear in the following views:

- Table view, which displays the trace data in a flat list.
- Outline view, which displays the trace data in a hierarchical list. Many instruments name the outline view the call tree view because it displays the call tree hierarchically.
- Diagram view, which displays a list of individual samples.
- Console, which displays output from your application.
- Source view, which displays source code.

The table, outline, and diagram views provide different views of the trace data. The table view shows the data in a flat list. The outline view shows the data in a hierarchical list, and the diagram view shows the data as a list of individual samples. Many instruments do not use the diagram view.

The table, outline, and diagram views all have different names that depend on the instrument. Use the jump bar to change views and to show the console.

Console

The console shows any command-line output your application generated when Instruments traced it. If you use `NSLog` or some other method to log debugging information, the logged output appears in the console.

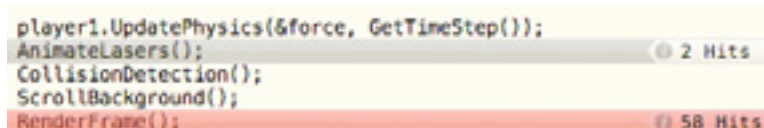
Source View

The table, outline, and diagram views display statistics at a high level. Showing statistics for a single function is as low as the table, outline, and diagram views go. The source view digs deeper. Suppose Instruments finds a problem in one of your functions, such as a memory leak or an excessive memory allocation. The source view helps you determine where the problem is coming from in your code.

The source view highlights interesting code. There are two ways to open the source view. First, double-click a function in the call stack in the extended detail view. Second, double-click a function in the call tree (outline view).

What the source view displays depends on the function you double-clicked. If you double-click a function you didn't write, the source view displays the function's assembly language code. If you double-click one of your functions, you will see the source code for that function. One or more lines of code in the function will be highlighted, as you can see in Figure 9.2. The highlighted code contains a message bubble that is similar to the message bubbles Xcode's editor displays for compiler errors. Instruments shows some relevant information for each highlighted line in the message bubble. The relevant information depends on the instrument. The Allocations instrument shows how much memory was allocated. The Time Profiler and Sampler instruments show the number of samples the instrument recorded. The relevant information may appear as a percentage instead of a value. If you have the extended detail view open, it lists the highlighted lines of code sorted by percentage, with the highest percentage lines of code appearing at the top of the list.

Next to the relevant information is an Info button. Clicking it opens a pop-up editor that shows the five heaviest backtraces or all the backtraces if there are less than five. What the heaviest backtraces represent depend on the instrument. If you're using the Leaks instrument, the five heaviest backtraces represent the five largest memory leaks. Use the arrow buttons to step through the backtraces. At the top of the pop-up editor is the relevant information for that backtrace, such as the amount of leaked memory.



```
player1.UpdatePhysics(&force, GetTimeStep());
AnimateLasers();
CollisionDetection();
ScrollBackground();
RenderFrame();
```

The screenshot shows a list of function calls in the Source View. The first four lines are in a light yellow background, and the last line, `RenderFrame();`, is highlighted in red. To the right of each line is a circular icon with a number and the word "Hits". The first four lines have a grey icon with "2 Hits", and the highlighted line has a red icon with "58 Hits".

Figure 9.2

Source view code

The source view has four buttons on the right side above the source code, which you can see in Figure 9.3. Some of these buttons are disabled if you're viewing code you didn't write. The first button is the Designate Source button. Clicking it opens an Open panel to find the source code file. The second button is the View Disassembly button. Clicking this button toggles showing your source code and disassembly, which consists of assembly language statements. Clicking the third button opens the file in Xcode. The last button lets you customize the display of statistics in the source view. You can display the statistics as a percentage or as a value. The percentages add up to 100% for a given function.

Xcode 4.4 adds a fifth button to the source view. To the right of the Designate Source button is a button that shows the source code and disassembly side by side.

Searching in the Detail View

Choose Edit > Find > Find to open the detail view's search bar. Click the magnifying glass icon in the search field to control where you search in the detail view. Where you can search depends on the instrument. Enter a search term in the search field and press the Return key to search.

When you search in the detail view, Instruments does not hide entries that don't meet the search criteria. Use the navigation buttons next to the search field to navigate the search results. If you want Instruments to hide entries that don't meet the search criteria, use the search field in the trace document window toolbar.

Extended Detail View

The extended detail view is not visible initially. Choose View > Extended Detail to show the extended detail view. What appears in the extended detail view depends on the instrument, but the call stack is what appears in most instruments.

Double-clicking a function in the call stack opens the function in the source view. For code you didn't write, all you see is the function's assembly language statements.



Figure 9.3

Source view buttons

Filtering Information

Instruments generates lots of trace data, which makes finding the important data difficult. Instruments has filtering features to help you find the important data.

Filtering by Time

In the trace document window's toolbar is the Inspection Range group, which contains a set of three inspection range buttons, shown in Figure 9.4. These buttons let you focus on samples that occurred over a period of time, which can help you determine the causes of memory leaks and excessive memory allocations.

The first step to using the inspection range buttons is to define the start of the inspection range. Click the timeline at the top of the track pane, and click the left inspection range button. Notice how everything after the starting point is highlighted in the graph.

After setting the start of the inspection range, you must set the end of the range. Click the timeline. Click the right inspection range button. Notice how the statistics in the detail view change to reflect the inspection range you set.

Clicking the center button erases the inspection range you set. Erasing the inspection range allows you to define a new range.

Searching

The trace document window toolbar has a search field. Enter a search term to filter the results in the detail view.

Clicking the magnifying glass icon in the search field lets you customize the filtering. When you click the icon, a menu opens that has two sections. The first section is instrument-specific. When you search a call tree, the first section lets you search for a symbol or a library. The second section determines how Instruments displays search results with multiple words: display results that match all the words you entered in the search field or display results that match any of the words you entered.

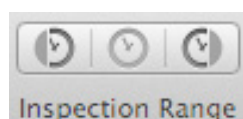


Figure 9.4

Inspection range buttons

Flagging Samples

A flag is a bookmark in a trace. Click the timeline where you want to add a flag. Choose Edit > Add Flag to add the flag. A blue flag marker appears on the timeline. Click the blue marker to open a pop-up editor that lets you name the flag, add a description, and change the flag color. Use the back and forward buttons in the pop-up editor to navigate the flags.

Flags you add are blue. Flags that Instruments sets have different flag marker colors, depending on the type of flag. If you create a heapshot with the Allocations instrument, Instruments marks the flag red.

To remove a flag, select it in the timeline and choose Edit > Remove Flag.

Instruments has a flag table that lets you manage the flags you set. Choose Window > Manage Flags to open it. The flag table lets you control the flags that appear in the track pane.

If you find yourself creating lots of flags, you can customize the toolbar to have a Flags group of buttons for adding and navigating flags. Right-click in the toolbar and choose Customize Toolbar.

Call Tree Data Mining

The call tree for a Mac or iOS application can be deep, which makes finding the information you want difficult. To the left of the detail view are controls to filter unwanted information from the call tree listing so you can focus on the information you need. There are three sections of controls: Call Tree, Call Tree Constraints, and Specific Data Mining, which you can see in Figure 9.5.

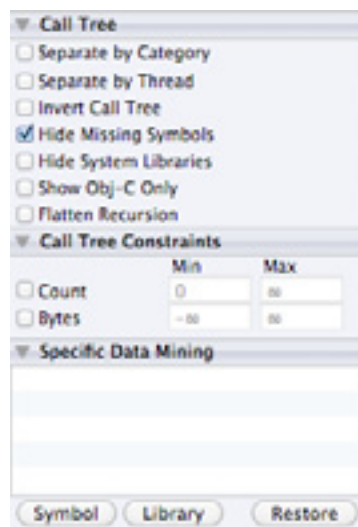


Figure 9.5

Call tree data mining

Make sure the instrument list is visible or you won't see the data mining controls. If you don't see the instrument list, choose View > Instruments to show it. Not every instrument has data mining controls.

You must be in the call tree view to enable the data mining controls. Choose Call Tree from the jump bar. When using call tree data mining, keep in mind that applying data mining to the call tree effects the statistics the call tree shows.

Call Tree Checkboxes

The Call Tree section contains the following checkboxes:

- Separate by Thread
- Invert Call Tree
- Hide Missing Symbols
- Hide System Libraries
- Show Obj-C Only
- Flatten Recursion

Some instruments have checkboxes to separate the listings by thread and by category. Separating by thread tells Instruments to show a call tree listing for each thread in your application. Separating by thread lets you focus on your application's main thread, which is where most of your application's activity takes place.

When Instruments shows the call tree, it starts at the top of the call tree. For C, C++, and Objective-C programs, the `main()` function is the top of the call tree. Inverting the call tree tells Instruments to start at the leaves of the call tree and work back to the root.

The values Instruments shows for a function in the call tree depend on whether or not you invert the call tree. Suppose you're using the Allocations instrument and it has a listing for a function, X. If you don't invert the call tree, the call tree tells you the amount of memory allocated where X is in the call stack. If you invert the call tree, the call tree tells you the amount of memory allocated where X is at the top of the call stack. Inverting the call tree lets you know how much memory the function X allocates. Inverting the call tree provides the most accurate information about a particular function.

When you hide missing symbols, Instruments hides functions that don't have symbols. When you hide system libraries, Instruments hides functions from system frameworks and libraries like Cocoa and Core Foundation. Hiding missing symbols and system libraries hide code you didn't write, allowing you to focus on your code.

Selecting the Show Obj-C Only checkbox tells Instruments to show only Objective-C functions in the call tree, hiding functions written in other languages. Showing only Objective-C functions can help you focus on your code, assuming your code is all Objective-C code.

A recursive function is a function that calls itself. If you select the Flatten Recursion checkbox, Instruments creates one listing for the recursive function.

Call Tree Constraints

The Call Tree Constraints section lets you use the values of the statistics to restrict what appears in the call tree view. Suppose you're using the Leaks instrument and you want to focus on leaks 1 KB and larger. Use the Call Tree Constraints section to limit the call tree display to symbols that leak 1 KB or more of memory.

Each constraint has a checkbox next to it along with Min and Max text fields. Select the checkbox next to a constraint to activate it. Enter a minimum value in the Min text field and a maximum value in the Max text field. If you want to view memory leaks 1 KB or larger, enter 1024 in the Min text field and leave the Max text field blank.

The constraints are instrument-dependent. The Leaks and Allocations instruments have a Bytes constraint that filters the call tree statistics on the size of the leak or allocation. Many instruments have a Count constraint that filters the statistics by the number of what Instruments recorded, such as number of leaks, number of memory allocations, and number of samples recorded.

Specific Data Mining

The Specific Data Mining section allows you to data mine a library or a symbol. To data mine a library, select a function from the call tree and click the Library button. When data mining a library you can charge the library to its callers or flatten the library to boundary frames. Charging a library to its callers hides the library's functions from the call tree, and its statistics are added to the functions that called the library's functions. Flattening a library excludes all the functions in a library except for its entry points. The costs of the excluded functions go to the entry points. A library's entry point is a function that is available to other programs.

To data mine a symbol, select a symbol from the call tree and click the Symbol button. When data mining a symbol you can charge the symbol to a caller or prune the symbol and its subtree. When you charge a symbol to its caller, the symbol is hidden in the detail view and its statistics are added to the function that called the charged symbol. When you prune a symbol and its subtree, that function and the functions it calls are hidden from the detail view.

Clicking the Restore button removes entries from the Specific Data Mining list. Deselect the checkbox next to a symbol or library to temporarily disable data mining for that symbol or library.

Data Mining Inside the Call Tree

Selecting a symbol from the call tree and right-clicking opens a contextual menu. The contextual menu has three sections. The top section has items to add entries to the Specific Data Mining section on the left side of the detail view. From the top section you can charge the symbol to its callers, prune the symbol from the call tree, charge the symbol's library to its callers, and flatten the library.

The center section of the contextual menu has items to focus on the selected symbol. You can focus on the following items for a symbol:

- The symbol's subtree. Focusing on a subtree focuses on the symbol and the subtree of functions the symbol calls. If you invert the call tree, the subtree is reversed, consisting of the subtree of functions that call the symbol.
- Calls made by the selected symbol, which shows the functions the symbol called.
- Callers of the selected symbol, which shows the functions that called the symbol.
- Calls made by the symbol's library, which shows the functions the symbol's library called.
- Callers of the symbol's library, which shows the functions that called functions in the symbol's library.

Focusing on a symbol makes that symbol the temporary root of the call tree. Use the jump bar to restore the call tree. Moving the mouse over a symbol makes a focus button appear. Clicking the button focuses on the subtree.

The bottom section of the contextual menu has one item. It reveals the symbol in Xcode.

Call Tree Tips

Option-clicking the disclosure triangle next to a symbol expands the entire call stack for the symbol. Expanding the call stack keeps you from having to click a bunch of disclosure triangles. Option-clicking a second time contracts the call stack.

For those of you working with the Cocoa or Cocoa Touch frameworks, selecting the Invert Call Tree and Hide System Libraries checkboxes can help you find your code in the call tree. When you find an interesting function in the call tree, focus on the subtree and deselect the Hide System Libraries checkbox so you can see the Cocoa or Cocoa Touch calls the function makes.

Run Browser

Choose View > Run Browser to open the run browser, which helps you manage multiple runs of a trace. The run browser is similar to the Cover Flow view in iTunes. The top half of the run browser has a list of runs. Select one from the list. You can name the selected run and enter a comment that describes what happened in the run.

Clicking the Promote Run button makes the selected run the current run in the trace document window. Clicking the Delete Run button deletes the run. Click the button in the lower right corner of the run browser to close the run browser and show the trace document window.

Exporting Trace Data

Choose Instrument > Export Track for InstrumentName to export the trace data to a CSV file that you can open in a text editor or spreadsheet. Not every instrument supports the export to CSV file feature.

Instrument-Specific Results

Now that I've discussed general techniques for examining trace results, it's time to look at the trace results from specific instruments. Because Instruments comes with over 30 instruments, there's no way to cover all of them in this book. I cover the following instruments: Leaks, Allocations, Time Profiler, OpenGL ES Analyzer, and Activity Monitor. The first four instruments are the most commonly used instruments. Covering the Activity Monitor instrument is the equivalent of covering five instruments: Activity Monitor, CPU Monitor, Disk Monitor, Memory Monitor, and Network Activity Monitor.

Leaks

The Leaks instrument reports any memory leaks it finds in your application. A memory leak occurs when your application allocates memory and doesn't free that memory. One small leak isn't a big deal, but large leaks and frequent leaks can cause problems. If you leak enough memory, your application will run slowly or crash. Eliminating memory leaks is especially important for iOS applications because iPhones and iPads have less memory than Macs.

Before You Trace

Before you check for memory leaks, you must make two decisions. First, do you want Instruments to automatically check for leaks? Second, do you want to gather the contents of leaked memory? Leaks is initially configured to automatically check for leaks but not configured to gather leaked memory contents.

On the left side of the detail view is a Snapshots section. Select the Automatic Snapshotting checkbox to automatically check for leaks. Use the Snapshot Interval text field to determine how often the automatic check occurs. The initial value is 10 seconds, which means Instruments checks for leaks every 10 seconds. Clicking the Snapshot Now button checks for leaks. If you deselect the Automatic Snapshotting checkbox, you must click the Snapshot Now button to check for leaks. You can use the Snapshot Now button even if you're automatically checking for leaks.

Selecting the Gather Leaked Memory Contents checkbox tells Instruments to gather the contents of leaked memory. The leaked memory's contents appear in the extended detail view below the stack trace. If the stack trace is deep, you must scroll past the stack trace to see the leaked memory contents. You must select a leaked memory block from the Leaks section to view the block's contents.

Leaked Blocks

When you check for memory leaks, Instruments initially shows the leaked memory blocks. Instruments provides the following information for each leaked block of memory:

- Leaked object. These are usually Cocoa classes, Core Foundation data structures, and general blocks of memory. General blocks of memory are indicated by an object name of Malloc.
- The number of leaks in the block. A blank entry means there is one leak in the block.
- Address, which is the starting address of the leaked memory.
- Size, which is the amount of leaked memory.
- Responsible library, which is the library where the memory leak was discovered.
- Responsible frame, which is the function where the memory leak was discovered.

If a block has multiple leaks, click the disclosure triangle next to the leaked object to see the individual leaks.

When you look at leaked blocks, it can be difficult to discover where in your code you're leaking memory. Suppose your code leaks a `NSString` object. The responsible library is going to be Foundation or AppKit, and the responsible frame is going to be a low-level `NSString` method. In this case the responsible library and responsible frame don't tell you the most important information: the location of the memory leak in your code. Open the

extended detail view to help discover the location of the memory leak. Selecting a leaked block shows the stack trace in the extended detail view. Showing the stack trace helps you find where the leak is coming from in your code.

History

Moving the mouse over a memory address makes a focus button appear. Clicking the focus button provides a history of memory-related events for the block of memory. Instruments shows the following information for each event:

- Category, which is usually the same as the Leaked Object column in the Leaks section.
- Event type. Common event types include Malloc, Free, Realloc, Release, Retain, and Autorelease.
- Timestamp, which is when the event occurred.
- RefCt, which represents the object's retain count.
- Address. The address will be the same for each entry because you're looking at the history of one block of memory.
- Size. Free events have a negative size. Retain, release, and autorelease events have a size of 0.
- Responsible library, which is the library that created the event.
- Responsible caller, which is the function that created the event.

One thing you'll notice when examining the history is the last entry has a retain count greater than 0. That is the mark of a memory leak. If the retain count were 0, the memory would be freed and there would be no leak.

Call Tree

Switching to the call tree view (outline view) can make finding the source of leaks easier. The call tree view initially tells you the total number of leaks and the total amount of leaked memory.

After you discover the total number of leaks and total amount of leaked memory, what you want to know is where you're leaking the memory. The checkboxes in the Call Tree section on the left side of the detail view can help you discover where the leaks are occurring in your code. Selecting the Invert Call Tree and Hide System Libraries checkboxes help you hone in on where the memory leaks are occurring in your code.

The call tree has three columns of information for each symbol in the call tree: Bytes Used, # Leaks, and Symbol Name. The Bytes Used column has two values: the amount of leaked memory and the percentage of leaked memory. The # Leaks column tells you the number of leaks. The Symbol Name column contains the name of the function and the library the function belongs to. The library for functions you write is the name of your application.

Double-clicking the name of one of your functions in the call tree opens the source view. The source view highlights the lines of code in the function where Instruments detected leaks. The highlighted lines of code are where you should start looking for memory-related problems.

Cycles and Roots

The Cycles and Roots section shows leak cycles. A leak cycle occurs when you lose a reference to a group of objects, which means the group of objects can no longer be reached. A leak cycle is worse than a single memory leak because you're leaking multiple objects. Checking for leak cycles helps fix memory problems in Objective-C code that uses automatic reference counting.

The first thing the Cycles and Roots section shows is the number of root leaks. A root leak can be a single leak or it can be the start of a leak cycle. Below the number of root leaks is a listing for each root leak. The Type column tells you the leaked object, and the Details column tells you additional information. For a leak cycle the Details column tells you whether you have a simple or complex leak cycle. Some leaks show the memory address of the leaked object in the Details column.

Some Type listings have a disclosure triangle next to them. The disclosure triangle indicates a leak cycle. Selecting a listing with a disclosure triangle fills the Graph column with a graph of the leak cycle. Hiding the extended detail view provides a better view of the graph. A red line in the graph indicates a strong reference. A blue line indicates manual reference counting. Double-clicking a line opens the code in Xcode if the leak cycle involves code you wrote. Clicking the disclosure triangle shows you the elements in the cycle.

Moving the mouse cursor over a listing's Type column makes a focus button appear next to the name of the leaked object. Clicking the focus button next to a root listing shows a history of memory events for that listing's memory address. This history shows the same information that I listed in the "History" section earlier in this chapter.

Track Pane

The track pane can also help you locate the source of memory leaks. The graph shows the number of leaks and the amount of leaked memory. A change in the graph means Instruments found memory leaks. Use the Inspection Range group of buttons in the toolbar to focus on the time when the leak was detected. The “Filtering by Time” section earlier in this chapter has more information on using the inspection range buttons.

Allocations

The Allocations instrument traces your program’s memory allocations. With this instrument you can see how much memory your program uses as well as examine every memory allocation your program makes.

Before You Trace

Click the Info button next to the Allocations instrument to open the pop-up editor. The pop-up editor for the Allocations instrument has two sets of checkboxes: Launch Configuration and Recorded Types. The Launch Configuration group has the following checkboxes:

- Discard unrecorded data upon stop. As its name indicates, selecting this checkbox tells Instruments to discard any unrecorded data when you click the Stop button.
- Record reference counts. Selecting this checkbox tells Instruments to record the reference counts of Objective-C objects.
- Only track active allocations. Deselecting this checkbox enables the Allocation Lifespan group of radio buttons on the left side of the detail view when you view the trace results.
- Identify C++ Objects. Selecting this checkbox makes your C++ classes appear as categories in the object summary.
- Enable NSZombie detection. Enabling NSZombie detection alerts you when you access a freed object in your application and helps you locate where you are accessing the freed object in your source code.

The Recorded Types group lets you control what the Allocations instrument records. It has the following checkboxes: Record All Types, Ignore Types with NS Prefixes, Ignore Types with CF Prefixes, and Ignore Types with Malloc Prefixes. Clicking the Configure button in the pop-up editor lets you add your own rules for recorded types.

Statistics

The Allocations instrument provides three levels of statistics about your program's memory allocations.

- Object summary
- Instances
- History

Object Summary

When you use the Allocations instrument, Instruments initially displays the object summary in the detail view. The object summary provides the following information for each memory allocation category:

- Graph. If the checkbox is selected, that category's memory allocations appear in the track pane's graph. The text color for the row changes to match its color in the graph.
- Category, which is usually a Cocoa class or a Core Foundation data structure. Many Core Foundation data structures are the C equivalents of Cocoa classes.
- Live bytes, which is the amount of memory currently allocated.
- # Living, which is the number of live memory allocations, the number of memory allocations that haven't be freed.
- # Transitory, which is the number of memory allocations that were freed. If you track only active allocations, the # Transitory column is 0.
- Overall bytes, which is the total amount of memory allocated while your program has been running.
- # Overall, which is the number of total memory allocations. It should equal the sum of the # Living and # Transitory columns.
- A histogram showing the ratio of live to overall memory allocations. The live allocations are in a darker shade than the overall allocations.

The first category in the object summary is All Allocations. The Live Bytes column for All Allocations is a good measure of the amount of memory your program is using at that moment in time.

A common category is Malloc. A Malloc category represents the allocation of a general block of memory that is not part of a Cocoa class or Core Foundation data structure. Allocations with a Malloc category have a number next to them. The number represents the number of bytes allocated. The entry Malloc 128 Bytes is a category consisting of all the 128-byte memory allocations your program made.

Let me use an example to show the relationship between the Live Bytes, # Living, # Transitory, Overall Bytes, and # Overall columns. Suppose your application makes 20 memory allocations of 1000 bytes each and frees the memory for 15 of the allocations. The Allocations instrument reports the following information for the 1000-byte memory allocations:

```
Live Bytes: 5000
# Living: 5
# Transitory: 15
Overall Bytes: 20000
# Overall: 20
```

The histogram colors range from blue to red, with purple and magenta as intermediate colors. The color indicates the ratio of live allocations (the # Living column) to total allocations (the # Overall column). Blue indicates a higher live to total allocation ratio. If you track only active allocations, the histograms are all blue because the ratio of live allocations to total allocations is 100% when tracking only active allocations. Red indicates a lower live to total allocation ratio. Red histogram bars signal a possible problem.

Instances

Moving the mouse cursor over a category in the object summary makes a focus button appear next to the category. Clicking the focus button shows the Instances section. The Instances section reports the following information about each memory allocation in that category:

- Address, which is the memory address where the memory was allocated.
- Category, which is the same as the category in the object summary.
- Timestamp, which measures when the allocation occurred relative to the start of recording.
- Live, which lets you know if an instance is live. A dot in the Live column signifies a live allocation. An allocation is live if it has not been freed.
- Responsible library, the library that made the request to allocate memory.
- Responsible caller, the function that made the request to allocate memory.

History

Moving the mouse cursor over an address in the Instances section makes a focus button appear next to the address. Clicking the focus button shows you the History section, which shows a history of every memory allocation event that happened at that memory address. The History section provides the following information for each memory allocation event:

- Category, which should be the same as the category in the object summary.
- Event type: Malloc, Free, Autorelease, Retain, Release, or Zombie. A zombie event occurs when you release an object whose retain count is zero.
- RefCt, which is the object's retain count.
- Timestamp, which is the same as the timestamp in the Instances section.
- Address. The address will be the same for each entry because all the allocations match the address in the Instances section.
- Size, which is the size of the memory allocation in bytes. Free events have a negative size.
- Responsible library, which is identical to the responsible library in the Instances section.
- Responsible caller, which is identical to the responsible caller in the Instances section.

To see a complete list of memory allocation events, you must tell the Allocations instrument to record reference counts, which I covered earlier in the “Before You Trace” section. If you deselect the Record reference counts checkbox, the History section shows only one event: the initial memory allocation.

Call Trees

The Statistics section shows you what your application allocates. Choosing Call Trees from the jump bar lets you see where your application allocates memory.

The call tree has three columns of information: Bytes Used, Count, and Symbol Name. The Bytes Used column has two values: size and percentage. The size is the amount of memory allocated. The percentage is the percentage of total allocation size.

The Count column tells you the number of memory allocations. The Symbol Name column tells you the name of the function and the library it belongs to. The library for functions you write is the name of your application.

Inverting the call tree shows paths where memory usage is high. Not inverting lets you see how much memory a function is allocating, both directly and indirectly through the functions the given function calls.

Double-clicking a function shows the percentage of memory allocations that come from the functions this function calls. If function A calls X, Y, and Z, you can see the percentage of memory allocated by X, Y, and Z.

Selecting the Separate By Category checkbox separates the call tree listings by memory category. Separating call tree listings by category helps you determine where in your code you're allocating something like an `NSString` object or a large Malloc category.

Objects List

Choosing Objects List from the jump bar lets you view every memory allocation. Select the All Objects Created radio button in the Allocation Lifespan section to see every memory allocation your application made. If you have the extended detail view open, you can view the call stack of every allocation as well as a description of each allocation. The description tells you the type of memory allocation event, the size of the allocation, and the retain count of the object being allocated. Select an allocation and use the arrow keys to step through each memory allocation.

The object list tells you the following information for each memory allocation:

- Address, which is the memory address where the memory was allocated.
- Category, which is usually Malloc, a Cocoa class, or a Core Foundation data structure.
- Timestamp, which measures when the allocation occurred relative to the start of recording.
- Live, which lets you know if an instance is live. A dot in the Live column signifies a live allocation. An allocation is live if it has not been freed.
- Size, which is the size of the memory allocation in bytes.
- Responsible library, which is the library that made the request to allocate memory.
- Responsible caller, which is the function that made the request to allocate memory.

Selecting the All Allocations category in the Object Summary and clicking the focus button displays the same information as the object list.

Heapshots

The Heapshots section lists the snapshots you took of your application's heap. Taking snapshots of the heap lets you measure heap growth and discover abandoned memory, which can help you discover hard to find memory leaks. Suppose you are writing a text editor and you want to see if creating a new document results in abandoned memory. Mark the heap, create a new document, close it, and mark the heap again. Creating a new document and closing it should result in no heap growth. The Heapshots section will tell you if the heap grew, and

if so, by how much. It will also tell you the objects that were allocated and are still in the heap. This is abandoned memory in my example. Heap growth does not necessarily mean you have abandoned memory; it depends on what you do between heapshots.

Click the Mark Heap button to take a snapshot of the heap. When you take a snapshot, Instruments adds a flag in the track pane's timeline. The Allocations instrument tells you the following information for each snapshot:

- Snapshot, which is the name of the snapshot.
- Timestamp, which tells you the time in the trace where you took the snapshot.
- Heap growth, which measures how much memory was allocated during the snapshot period.
- # Persistent, which is the number of persistent memory allocations that occurred during the snapshot period.

When you take a snapshot, Instruments gives it a name like Heapshot 1. Double-click the name to give the snapshot a more meaningful name that describes what the heapshot is supposed to measure.

Clicking the focus button next to a snapshot switches to the call tree view, where you can look at your code to see what is causing the memory increase. The call tree view for a snapshot is the same as the call tree view for normal memory allocations.

Clicking the disclosure triangle next to a snapshot shows the categories of memory allocations that occurred in the snapshot. It provides the following information for each category:

- Category, which is similar to the Category column in the Statistics section.
- Heap growth, which is the amount the heap grew because of the category.
- # Persistent, which is the number of persistent memory allocations.

Clicking the disclosure triangle next to a category shows the memory address of each instance along with its timestamp and size. Clicking the focus button next to an instance shows the same columns of information in a heapshot as it does for normal memory allocations: Address, Category, Timestamp, Live, Size, Responsible Library, and Responsible Caller.

Clicking the focus button next to a memory address shows a history of memory allocations for that address. The history for a heapshot shows the same columns of information as the history of a normal memory allocation: Category, Event Type, RefCt, Timestamp, Address, Size, Responsible Library, and Responsible Caller.

Time Profiler

The Time Profiler instrument measures where your application spends its execution time. It records the call stack periodically (every millisecond initially). The Sampler instrument records similar information. The Time Profiler instrument has lower overhead than the Sampler instrument. In most cases you should use the Time Profiler instrument.

You might be wondering how recording the call stack determines where your application spends its time. The function at the top of the call stack is the function that was executing when the Time Profiler instrument recorded the call stack. If a function appears at the top of the call stack in lots of samples, you know your program spent a lot of time in that function. When a program runs slowly there are a few areas of code where the program spends most of its time. Recording the call stack allows you to find the slow areas in your code without requiring any additional coding. Profile your program with the Time Profiler instrument.

Call Tree

When you record a trace with the Time Profiler instrument, Instruments shows the call tree in the detail view. The call tree for the Time Profiler instrument has three columns of information: Running Time, Self, and Symbol Name. The Symbol Name column tells you the name of the function. Next to the function is the library or framework where the function resides. For code you wrote, you will see the name of your application instead of a library name.

The Running Time column has two values: an amount of time and a percentage. The time value is the amount of time (in milliseconds) the symbol appears on the call stack. The percentage is the percentage of samples the symbol appears on the call stack. The amount of time a function appears on the call stack doesn't necessarily mean much. If you're writing a Cocoa application in Objective-C, your `main()` function is going to have a high running number because it is on the call stack the whole time your program is running. The `main()` function does very little in Objective-C applications. Optimizing the `main()` function isn't going to make your application run faster.

The Self column tells you the amount of time the function appears at the top of the call stack. When a function appears at the top of the call stack, your application was in that function when Instruments recorded the sample. A function with a high Self value is a prime target for optimization. If you invert the call tree, the Self column matches the amount of time in the Running Time column. If you don't invert the call tree, the Self column tells you the amount of time your application spent in the function.

If you're running an older version of Xcode, you won't have a Self column. The information that would be in the Self column is in the Running Time column. The Running Time column will show two percentages for a function. The Self information is the percentage in parentheses.

When the call tree displays a percentage for a particular function, the displayed percentage is the percentage of samples that are currently visible in the call tree view. The percentage can change depending on your actions. Actions like inverting the call tree, hiding system libraries, choosing a thread to focus on, and focusing on a function can change the percentages Instruments displays.

If you have the extended detail view open, it lists the heaviest stack trace when you select a symbol. Each function in the stack trace has a number next to it. This number represents the number of samples where the function appears on the call stack.

Finding Heavy Paths

Inverting the call tree makes it easier to find where your program is spending its time. But if you code with Apple technologies, the leaves of the call tree tend to be low-level system calls, which makes finding your code difficult. Selecting the Hide System Libraries checkbox makes finding the slow parts of your code easier. You also may want to select the Separate By Thread checkbox.

When you invert the call tree, each listing in the call tree view represents a path from that function, the leaf of the tree, to the root. If you have a listing for a function and the Self value is 500 milliseconds, it means your application spent 500 ms in this path. It does not necessarily mean your application spent 500 ms in this function. The same function can appear in multiple paths.

Focusing on a Subtree

Sometimes when you're profiling, there's a certain section of code you're interested in. Suppose you're profiling a game. Games spend most of their time in a game loop, where you read player input, move the player, move the player's opponents, and draw the scene. When profiling a game, the game loop is the most important area of code to examine.

The solution is to focus on your `GameLoop()` function. Deselect the Invert Call Tree checkbox. Find the `GameLoop()` function in the call tree. Click the focus button next to `GameLoop()` to focus on it, making `GameLoop()` the temporary root of the call tree.

Finding Where a Function Spends Its Time

Double-clicking a function in the extended detail view or in the call tree opens the function's source code file in the source view. The source view can look at a function call and show you the lines of code in that function where the Time Profiler instrument recorded a sample. If the Time Profiler instrument recorded a sample in a line of code, that line has an Info button that shows you the heaviest backtraces. When you look at the source code, the extended detail view shows the functions the current function called, sorted by the percentage of samples recorded.

Suppose you have a function, function A, that calls functions X, Y, and Z. Double-clicking function A tells you the percentage of time it spent in X, Y, and Z.

Sample List

If you choose Sample List from the jump bar, you can see each sample Instruments took. Instruments displays the following information for each sample:

- Sample number.
- Timestamp, which is when Instruments took the sample, relative to the start of the trace.
- Depth, which is the stack depth.
- CPU, which is the CPU core where Instruments took the sample. Some Macs support hyperthreading, which means each physical CPU core has two logical cores that can execute a thread. My Mac has a dual-core processor that can handle four threads at once. On my Mac the CPU column ranges from 0-3. If your Mac's processor can handle more simultaneous threads, you will have a higher range of values. This field may be blank for iOS applications.
- Thread, which is the thread where Instruments took the sample.
- Process, which should be the name of your application.
- Hot frame, which is the function that was encountered most frequently during sampling.
- Responsible library, which is the module that owns the hot frame.
- Responsible caller. Many times the responsible caller is the same as the hot frame.

If you have the extended detail view open, you can walk through the call stacks.

When you look at the sample list initially, you may have a difficult time sifting through the data. The Time Profiler instrument takes many samples. The hot frame for most samples is a function you didn't write, which makes finding your code difficult. The search field in the detail view can help you find your code in the sample list. Choose Edit > Find > Find to open the search field.

Click the magnifying glass icon in the search field to pick the categories on which to search. Searching the responsible library and entering your application name lets you see the samples where your application is the responsible library. Entering a class name for the responsible caller lets you see the samples where that class is the responsible caller. Entering a class name for the backtrace lets you see the samples where the class appears in the stack trace. Searching in the detail view can be confusing because Instruments does not hide the samples that don't meet the search criteria. Use the navigation buttons next to the search field to navigate the search results.

Selecting an item from the list of samples in the Time Profiler instrument and clicking the left inspection range button hides the samples that occurred before the selected sample. Clicking the right inspection range button hides the samples that occurred after the selected sample. Clicking the center inspection button restores all the samples.

Strategy Bar

Instruments adds a strategy bar to the Time Profiler instrument. The strategy bar, which you can see in Figure 9.6, is above the track pane. On the left side of the strategy bar are three small buttons. The center one is initially selected, which is the Instruments strategy that displays CPU activity in a single track. The Instruments strategy is no different from the track pane of any of the other instruments.

Clicking the left button shows the CPU strategy, which creates a track pane for each CPU core. The track panes display CPU activity for each core. The CPU strategy lets you see how well your application uses each core.

Clicking the right button shows the threads strategy, which creates a track pane for each thread. The threads strategy displays CPU activity for each thread.

Next to the strategy bar's strategy buttons are a series of menus. Use the menus to view specific CPU cores, processes, and threads. The rightmost menu applies to the CPU and threads strategies. For the threads strategy the menu lets you choose what appears in the track pane: CPU usage or callstack samples. If you choose Callstack Samples, you can click on a sample in the track pane to view the call stack Instruments recorded for that sample.



Figure 9.6

Strategy bar

For the CPU strategy this menu controls how Instruments graphs CPU usage in the track pane. You can chart by usage, boundary, and category. Charting by usage shows total CPU usage. Charting by boundary divides the CPU usage between kernel code and user code. Charting by category divides CPU usage by category, such as application, graphics, audio, and networking.

If you chart CPU usage by boundary or category, the button on the right side of the strategy bar lets you customize the colors Instruments uses in the track pane. Click the button to see a list of boundaries or categories. Click the color next to a boundary or category to change its color in the track pane.

OpenGL ES Analyzer

The OpenGL ES Analyzer instrument analyzes your OpenGL ES application for correctness and performance problems. In addition to pointing out problems in your OpenGL ES code, the OpenGL ES Analyzer instrument provides recommendations for fixing the problems. The OpenGL ES Analyzer instrument has the following sections:

- Expert
- Frame Statistics
- Trace
- Call Trees
- API Statistics

Expert

The Expert section alerts you to OpenGL ES-related problems. It displays the following information for each problem the instrument finds:

- Severity, which indicates how severe the problem is.
- Total occurrences, which is the number of times Instruments encountered the problem during the trace.
- Unique occurrences, which is the number of places in your code where the problem occurred.
- Category, which is the type of problem. An example of a problem category is a redundant OpenGL ES call.
- Summary, which provides a brief summary of the problem.

The OpenGL ES Analyzer instrument uses color to indicate the severity of the problem. Orange indicates medium severity, which means the problem is worth investigating to see if it needs to be addressed. Red indicates high severity, which means the problem most likely needs to be addressed. Problems with red severity generally have incorrect behavior or cause performance problems.

The difference between total occurrences and unique occurrences deserves additional explanation. Suppose you have a problem with a redundant call to `glEnable(GL_TEXTURE_2D)` in one of your functions. Your code calls the function with the redundant `glEnable()` call 50 times. In this case Instruments reports 50 total occurrences and one unique occurrence.

If you have the extended detail view open, selecting a problem shows an explanation of the problem with possible solutions as well as a stack trace.

Moving the mouse cursor over a problem category shows a focus button. Clicking the focus button gives you one listing for each unique occurrence of the category. It tells you the following information for each unique occurrence:

- Severity.
- Total occurrences.
- Number of commands involved, which is the number of OpenGL ES commands involved in the problem.
- Responsible command, which is the OpenGL ES command that caused the problem.

Clicking the focus button next to the number of commands involved or the responsible command opens a trace of OpenGL ES calls for the selected command. OpenGL ES commands that are involved with the problem are highlighted with black text in the trace. If you have the extended detail view open, you can see a stack trace for each OpenGL ES command.

Frame Statistics

The Frame Statistics section has one listing for each OpenGL ES frame in the trace. There is a column in the table for each statistic you told the OpenGL ES Analyzer instrument to list. You can list the following statistics:

- # Batches
- # Disables
- # Enables
- # Flushes
- # GL and platform calls
- # GL calls
- # Lines rendered

- # Platform calls
- # Points rendered
- # Redundant state changes
- # Render passes
- # Texture loads
- # Texture uploads
- # Triangles rendered
- Uploaded texture bytes

Each of these statistics can be displayed in the track pane. Click the Info button next to the OpenGL ES Analyzer instrument in the instrument list to control the statistics that appear in the track pane.

Trace

The Trace section lists every OpenGL ES function call your application made during the trace. If you have the extended detail view open, you can see a stack trace for each function call in the trace.

Call Trees

The Call Trees section has the following columns of information for each symbol in the call tree: Running Time, Running Count, and Symbol Name. The Symbol Name column tells you the name of the function. Next to the function is the library or framework where the function resides. For code you wrote, you will see the name of your application instead of a library name.

The Running Time column has two values: an amount of time and a percentage. The amount of time is the amount of time the function appears on the call stack. The percentage is the percentage of total time the function appears on the call stack.

The Running Count column has two values: a count and a percentage. The count is the number of samples Instruments took where the symbol appears on the call stack. The percentage is the percentage of samples where the symbol appears on the call stack.

When examining the call tree in the OpenGL ES Analyzer instrument, you should hide missing symbols instead of hiding system libraries. OpenGL ES is a system library, and you don't want to hide OpenGL ES calls when analyzing OpenGL ES performance in your application.

The information displayed in the call tree view for the OpenGL ES Analyzer instrument is similar to the information displayed in the Time Profiler instrument. Read the “Time Profiler” section in this chapter for more information on what you can do in the call tree view.

API Statistics

The API Statistics section has one listing for each OpenGL ES function your program called. It lists the following information for each function:

- The function name.
- Count, the number of times it was called.
- The total time (in microseconds) your application spent in the function.
- Average time (in microseconds), which is the total time divided by count.

Single Frame Navigation

The left side of the detail view has a Frame Navigation section. Clicking the Single frame navigation checkbox tells Instruments to show the frame statistics, call tree statistics, and API statistics for one frame instead of showing the statistics for all frames. Single frame navigation allows you to view changes in the track pane from frame to frame, showing the differences that occur in a frame. Use single frame navigation to focus on one frame at a time.

Use the Back and Forward buttons under the Single frame navigation checkbox to move to the next or previous frame. Enter a frame number in the text field to move to a specific frame.

Overriding the Pipeline

The left side of the detail view has an Overrides section. You can use this section to override the OpenGL ES pipeline. Why would you want to override the pipeline? You may want to disable parts of the pipeline to determine if those parts of the pipeline are the bottlenecks in your application.

You must start tracing your application to access the Overrides section. Click the Pause button to override the pipeline. Overriding the pipeline can mess up the look of your application. Instruments warns you of the dangers of overriding the pipeline when you override part of the pipeline.

Activity Monitor

The Activity Monitor instrument records system activity. This instrument records all the information the CPU Monitor, Disk Monitor, Memory Monitor, and Network Activity Monitor instruments record. That is why I am focusing on the Activity Monitor instrument. The main difference between the various Monitor instruments is what they are initially configured to graph in the track pane. But you can configure the Activity Monitor instrument to graph additional information by clicking the Info button next to the Activity Monitor instrument in the instruments list.

Summary

The Activity Monitor instrument initially shows the summary. The summary lists the following information for each running process:

- Process ID.
- Process name.
- User name, which for most processes is your username or `root`.
- % CPU, which is the percentage of CPU time the process used during the trace.
- Threads, which is the number of threads the process has.
- Real memory, which is the amount of RAM the process is using.
- Virtual memory, which is the amount of virtual memory the process is using.
- Architecture, which is Intel, PowerPC, or ARM. It also tells you if the process is running in 64-bit mode.
- CPU time, which is the total amount of CPU time the application has used since you launched it.
- Sudden termination, which tells you the process's support for sudden termination. Sudden termination applies only to Mac processes.

The percentage of CPU time and real memory are the pieces of information that application developers care about most.

Sudden termination requires some additional explanation. Apple introduced sudden termination in Mac OS X 10.6 to make system shutdown faster. On earlier versions of Mac OS X, the operating system sends a quit application event to each running application when you shut down your Mac. With sudden termination the operating system kills any running application that supports sudden termination and isn't in a dirty state. An application in a dirty state is one that has some cleaning up to do or has data to save when quitting the application. An example of a dirty application is one with a document that has unsaved changes. The operating system won't kill the application in this case because the application has to ask the user to save the changes.

If a process does not support sudden termination, its Sudden Termination column says N/A. A process that supports sudden termination has either Yes or No in the Sudden Termination column. Yes means the application can be killed by the operating system in its current state. No means the application can't be killed because its disable count is greater than zero. The disable count is in parentheses.

Parent Child

The Parent Child section shows the same information as the Summary section but uses a hierarchical process listing. The flat process listing should be sufficient for most of you, but the hierarchical process listing is available for those who need it.

Samples

Switching to the diagram view shows total system activity statistics rather than statistics for a single process. It lists every statistic the Activity Monitor instrument can graph in the track pane. The Samples section shows you the following information for each sample:

- Total threads
- Physical memory wired
- Physical memory active
- Physical memory inactive
- Physical memory used (equals sum of wired, active, and inactive physical memory)
- Physical memory free (this plus physical memory used equals total RAM on your Mac)
- VM size
- Page ins
- Page outs
- Swap used
- Net packets in
- Net bytes in
- Net packets out
- Net bytes out
- Net packets in per second
- Net packets out per second
- Net bytes in per second
- Net bytes out per second
- Disk read ops
- Disk bytes read
- Disk write ops
- Disk bytes written
- Disk read ops per second

- Disk write ops per second
- Disk bytes read per second
- Disk bytes written per second
- % Total load (equals user load + system load)
- % User load
- % System load

Trace Highlights

The Activity Monitor instrument has a Trace Highlights section, which has graphs for CPU%, CPU time, and real memory usage. It lists the top 5 processes for each category. For CPU% it lists the five processes that used the highest percentage of the CPU.

To get to the Trace Highlights section, use the menu on the left side of the jump bar. The menu's text should be Activity Monitor.

Creating a Custom Instrument

A cool feature of Instruments is that you can create your own instruments. To create a custom instrument, choose Instrument > Build New Instrument. You must have a trace document open to create an instrument. The instrument you create will be added to the trace document.

Parts of an Instrument

An instrument has the following contents:

- Name, category, and description
- A DATA section
- A BEGIN probe
- One or more probes
- An END probe

The name, category, and description identify the custom instrument in Instruments so you can find it in the Library.

The DATA section contains any global data you want to use in the instrument. If you're going to use a variable in multiple probes, place the variable in the DATA section.

The BEGIN probe contains any initialization code you want to supply. The END probe contains any code you want to execute when the trace ends.

The probes are the heart of an instrument. Probes are similar to functions in an application. An application consists of a collection of functions, and an instrument consists of a collection of probes.

You can also record a stack trace for your instrument. You can record a user stack trace, a kernel stack trace, or a Java stack trace. Most applications should record a user stack trace. Those of you writing an instrument for Java should record a Java stack trace.

Parts of a Probe

A probe has two parts: a provider and an action. The provider determines when the probe should fire. The action determines what the probe does when it fires. Normally the action consists of executing a DTrace script and recording data.

Determining When the Probe Fires

When you build a new instrument, Instruments supplies an empty probe for you, which you can see in Figure 9.7. Instruments gives the probe the name Probe 1. You can enter a more descriptive name in the text field. Next to the text field is a menu where you choose the provider. Table 9.1 contains a list of probe providers.

After choosing a provider, you must specify additional information. What you specify depends on the provider. If you pick the Objective-C provider, you must supply a class name and a method name as well as choose whether to fire on entry or exit. If you choose the Core Data provider, pick an event from the menu.

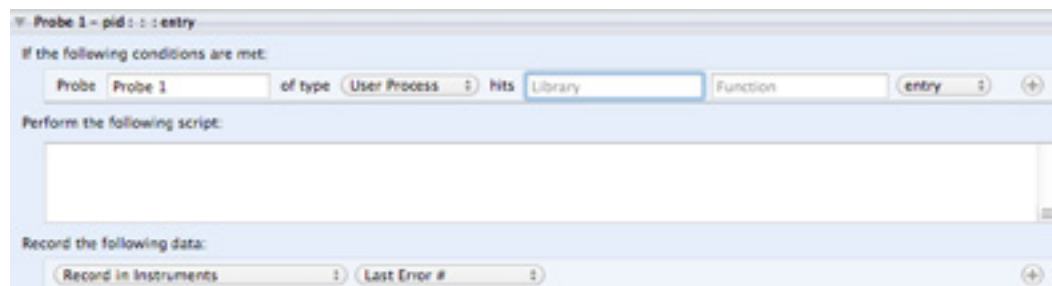


Figure 9.7

Empty probe Instruments supplies for a custom instrument

Click the + button on the right side to add predicates to your probe. Adding predicates is optional. When you add a predicate, you should see three menus and a text field. The left menu contains a list of data to inspect, which consists mostly of DTrace variables, but you can choose Custom and supply an expression. The second menu contains a list of comparison operators: equal, not equal, greater than, less than, greater than or equal, and less than or equal. Enter a value in the text field. Use the third menu to create multiple predicates. Use the + button to add another predicate. The third menu lets you AND or OR the predicates.

Table 9.1 Probe Providers

Provider	Description
User Process	Record when you enter or exit a function in a specific library.
Objective-C	Record when you enter or exit a method in an Objective-C class.
System Call	Record when you enter or exit a system call.
DTrace	Record when your instrument enters one of the following DTrace probes: BEGIN, END, or ERROR. An ERROR probe fires when a run-time error occurs during tracing.
Kernel Function Boundaries	Record when you enter or exit a kernel function.
Mach	Record when you enter or exit a Mach library function.
Profile	Record at a specified time interval on each CPU core.
Tick	Record at a specified time interval on a single CPU core.
I/O	Record when you enter a kernel I/O routine.
Kernel Process	Record when you call a kernel-level routine.
User-Level Synchronization	Record when entering a synchronization function that prevents thread conflicts.
CPU Scheduling	Record when you turn a CPU core on or off.
Core Data	Record when a low-level Core Data event begins or ends.
Custom	Use a custom provider. Supply a provider, module, function, and name. Creating custom providers is an advanced topic that is beyond the scope of this book.

Performing the Action

Instruments provides a text view to enter your script. Write your script in D, which is a scripting language for DTrace. Don't confuse DTrace's D language with the D programming language. They both have the name D, but they're two different languages. Explaining the D scripting language is beyond the scope of this book, but if you have programmed in C or C++, you should be able to grasp D. Oracle's DTrace site has extensive information on writing DTrace scripts. I have a link to Oracle's DTrace site on the *Xcode Tools Sensei* Resources page. After writing your script you must tell Instruments what to record. You don't have to record data for every probe, but you should record data for at least one probe. If none of your probes record data, there is little point in creating a custom instrument.

When recording there are two menus. The first menu has choices to record or not record. If you choose not to record any data, the second menu disappears. The second menu tells Instruments what to record. Your recording choices mostly consist of DTrace variables. Some examples of DTrace variables are the timestamp, stack depth, CPU core number, and the name of the current function. Choose Custom to record one of the variables in your custom instrument. If you choose Custom, you can choose the data type to display the variable: string, character, integer, unsigned long long, float, or hex.

Custom Instrument Example

Creating an instrument is a complicated topic. The best way to explain it is with an example. The custom instrument in this example measures the amount of time spent in the Objective-C method `readFromFileWrapper:`. You should be able to modify this example to measure the time spent in one of your application's methods.

This example consists of two probes. The first probe records the timestamp when the application enters the `readFromFileWrapper:` method. The second probe records the timestamp when the application exits the method. Subtract the start time from the end time to measure the amount of time spent reading a file wrapper.

Start

Create a blank trace document (choose File > New) and choose Instrument > Build New Instrument. Your new instrument's initial name is DTrace Instrument. Give it a more meaningful name. You'll also want to give the instrument a better description than DTrace Instrument. The instrument will go in the category Custom Instruments, which is good for now. If you find yourself creating lots of instruments, you can create new categories for your custom instruments.

DATA

Our instrument needs three variables: the start time, the end time, and the elapsed time. The timestamp Instruments uses is a 64-bit integer so the variables must be 64-bit integers. Click the disclosure triangle next to DATA and enter the following code:

```
int64_t startTime;
int64_t endTime;
int64_t elapsedTime;
```

BEGIN

When the instrument starts, we should initialize the three variables to zero. Click the disclosure triangle next to BEGIN and enter the following code:

```
startTime = 0;
endTime = 0;
elapsedTime = 0;
```

First Probe

The Probe text field has the value Probe 1. You can enter a more descriptive name if you want. There is a type pop-up menu whose initial value is User Process. Choose Objective-C from the type menu. After choosing Objective-C you should see a class text field, a hits text field, and a menu whose initial value is entry.

Enter the name of your class in the class text field. I'm using the class `Book`. Enter the name of the method in the hits text field. To record the `readFromFileWrapper:` method, enter the following text:

```
-readFromFileWrapper?ofType?error?
```

The `?` character separates the arguments an Objective-C method takes.

Now we have to provide a script. The script should set the variable `startTime` to the current timestamp value. Enter the following code in the text view:

```
startTime = timestamp;
```

The variable `timestamp` is part of DTrace. It contains the current value of the CPU's timestamp counter.

We're going to record the timestamp value to make sure the custom instrument works properly. Choose Relative Timestamp from the pop-up menu. You could also have told Instruments to store a custom value, `startTime`. The variable `startTime` is the same as the timestamp; you could go either way. Recording the timestamp is easier.

Second Probe

Click the + button in the lower left corner of the window to add a second probe. Choose Objective-C for the provider type. Enter the same class name in the class text field that you entered for the first probe. Enter the same method name in the hits text field that you entered for the first probe. Choose return from the menu to the right of the hits text field.

In the script we set the variable `endTime` to the current timestamp value. Then we calculate the elapsed time by subtracting `startTime` from `endTime`.

```
endTime = timestamp;  
elapsedTime = endTime - startTime;
```

We'll tell Instruments to record the second timestamp value and the elapsed time. Choose Relative Timestamp from the pop-up menu. Click the + button on the right side to add another variable to record. Choose Custom from the pop-up menu. There will be two text fields next to the menu. The first text field contains the variable you want to record; enter the value `elapsedTime`. The second text field contains the display name, which will be the column heading that appears in the detail view when you perform a trace. Enter **Elapsed Time** or whatever you want. To the right of the text fields is another pop-up menu that specifies the data type of the variable being displayed. Choose Unsigned Long Long because we're displaying a 64-bit integer value.

Click the Save button to finish creating the instrument. The instrument you created should be added to your trace document.

Running a Trace

After saving your custom instrument choose an application to trace and click the Record button to start recording. Instruments will display the timestamps and elapsed time. It may also display a Caller column, which is the function that made the call to the method being traced.

Improvements to the Custom Instrument

If you want to try making custom instruments, I can think of three improvements you can make to this example. First, the instrument in the example measures the elapsed time in nanoseconds. You could modify the instrument to display the elapsed time in seconds or milliseconds.

Second, you might care only about the elapsed time, not the timestamps for the two probes. Click the minus button next to the pop-up menu with the value Relative Timestamp to stop recording the timestamps.

Third, you could improve the instrument by keeping a running total of the time spent in the method you're tracing. The instrument currently records the time spent in a single method call. Recording the total time would help.

Editing an Instrument

To edit a custom instrument, create a trace document and add the instrument to the trace document. Select the instrument from the instrument list and choose Instrument > Edit 'InstrumentName' Instrument. Some of the built-in instruments can be edited. If a built-in instrument is editable, the Edit Instrument menu item is enabled in the Instrument menu.

When you choose to edit an instrument, the build instrument sheet opens for you to make changes. Click the Save Copy button when you're done making changes. Clicking the Save Copy button adds a copy of the instrument to the Library. I recommend changing the instrument's name or description when editing an instrument. If you don't change the name or description, you end up with two copies of the instrument in the Library with the same name and description. Determining which instrument is the updated one can be difficult.

Chapter 10

Command-Line Debugging Tools

Over the years Unix programmers have created many command-line tools to help them find problems in their code. Because Mac OS X has Unix at its core, you have access to all these tools as well. I do not recommend starting with the command-line tools. You will find it much easier initially to use Xcode's debugger or Instruments to find problems in your code. When you need to dig deeper to find a nasty bug, try the tools in this chapter.

A Command Line Primer

For those of you who have used only graphical user interfaces, the Unix command line in Mac OS X may seem intimidating, but this section introduces you to enough of the Unix environment for you to use the tools covered in this chapter. To reach the command line, run the Terminal application. It should be in your Applications folder under Utilities. After launching Terminal a Unix shell window opens, and you can start working in the command line.

Executing Commands as root

Several of the tools I cover in this chapter require you to run them as the `root` superuser, but you don't have to switch to `root`. The `sudo` command allows you to execute any Unix command as though you were `root`. Supply the command you want to execute or the program you want to run.

```
sudo CommandName
```

After running the `sudo` command you will be prompted for a password. Enter the password for your user name. The command executes after you enter your password.

Navigating Directories

If your Mac is like mine, it has thousands of directories (folders) for applications, source code, software development kits, and programming documentation. Navigating these directories is more difficult to do from the command line than from the Finder. Two UNIX commands help you navigate your computer's directories: `ls` and `cd`.

The `ls` command lists the files and subdirectories inside the current directory. Without the `ls` command you would have to memorize the contents of your computer's folders to do any navigation.

The `cd` command changes the current directory. To change to a subdirectory of the current directory, enter the subdirectory name. Suppose the current directory has a subdirectory **Games** that holds your favorite games. If you wanted to move to the **Games** directory, you would enter the following command:

```
cd Games
```

To move through several levels of subdirectories at once, use slashes to separate the subdirectories. Suppose you wanted to move to the directory where you have the game **Starcraft**. You could execute the `cd` command twice, once to move to the **Games** directory and once to move to the **Starcraft** directory. Or you could move directly to the **Starcraft** directory with one `cd` command.

```
cd Games/Starcraft
```

To move back one directory, use `..` as the directory name. If you were in the **Starcraft** directory and wanted to move back to the **Games** directory, you would enter the following command:

```
cd ..
```

If you have a leading slash in the directory name, the directory moves relative to the hard disk on which you installed Mac OS X. To move to your Applications directory, you would enter the following command:

```
cd /Applications
```

If one of your directories has a space in it, put the directory name in quotes.

```
cd "Source Code"
```

Without the quotes, the operating system treats the directory name **Source Code** as two separate arguments, and you will get an error message "Too many arguments".

Getting Help

If you get stuck in the command line, Unix manual pages are your friends. There are manual pages for virtually every Unix system command and command-line tool. They describe what the command (or tool) does, the arguments it takes, and the available options. To read a manual page, use the `man` command.

```
man CommandName
```

To learn about the options available for the `ls` command, enter the following command:

```
man ls
```

Most manual pages are too large to fit in a shell window so only the first part of the manual page appears in your window. The bottom line of your shell window will have a colon with the cursor next to it to tell you there's more text to read. To see the next line of text, press the down arrow key or the Return key. To see the next page (the amount of text that will fit in the window) of text, press the space bar. Press the up arrow key to see previously viewed parts of the manual page that are no longer visible in the window.

Finding Your Application's Process ID

Some of the tools in this chapter require you to supply your application's process ID. Every running application in Mac OS X has a process ID, which is a number that uniquely identifies the application to the operating system. You have no way of knowing your application's process ID until you launch it.

To find your program's process ID, open a new shell window by choosing `Shell > New Window > Basic`. Run the `top` tool by typing the word `top`. `top` lists all the currently running processes with information about each process, such as how much CPU time and memory they are using. When you run `top`, look at the left side of the shell window to see a column with the heading `PID`. This column lists each running application's process ID. Find your application's process ID and be ready to enter it when running the debugging tools in this chapter.

fs_usage

The `fs_usage` tool presents a real-time listing of file system calls and page faults. Before running `fs_usage`, make your shell window as wide as possible; the wider the window, the more information `fs_usage` returns. You must use the `sudo` command to run `fs_usage`.

Running `fs_usage`

If you supply no arguments to `fs_usage`,

```
sudo fs_usage
```

It displays information on all running processes. Running `fs_usage` on all running processes is usually a bad idea. `fs_usage` displays each file system call as it occurs, and Mac OS X applications make a surprisingly large number of file system calls, making it hard to keep up with all the information `fs_usage` reports. In most cases the only process you're interested in is your application. To limit `fs_usage`'s reporting to your application, supply either the application's process ID or its name.

```
sudo fs_usage ProcessID
sudo fs_usage AppName
```

When you run `fs_usage` on your application, the shell window seems to lock up with no data appearing in it. The lack of activity reflects the fact that you're in the Terminal application, which means your application isn't doing anything at the moment. Switch back to your application, do some things in it, then switch back to Terminal. Your shell window should be packed with data from `fs_usage`.

What `fs_usage` Tells You

Assuming you made your window wide enough, `fs_usage` provides the following data for each system call:

- A timestamp of when the system call occurred.
- The name of the system call. Table 10.1 lists the most common system calls.
- Additional information that depends on the nature of the system call. Make sure your shell window is wide or the additional information will not appear. Look at Table 10.2 for the types of additional information `fs_usage` provides.
- The pathname of the file the system call accessed.
- The amount of time, in seconds, the program spent in the system call. If the listing has a W next to it, the amount of time includes the time spent waiting for a file operation to finish.
- The name of the process that made the system call. The process name is useful only if you're viewing multiple processes with `fs_usage`.

Table 10.1 Common System Calls

System Call	Description
CACHE_HIT	The computer found the data it needs inside one of its caches. Cache hits are good because the computer doesn't have to go to RAM or disk to retrieve the data.
PAGE_IN	The operating system moved data from disk to memory.
PAGE_OUT	The operating system moved data from memory to disk.
read	Read data from a file.
write	Write data to a file.
open	Open a file.
close	Close a file.
lseek	Move to a particular place in a file.
fstat	Retrieve information about an open file, such as its size, the last time it was accessed, and the last time its contents changed.
stat	Retrieves the same information as <code>fstat</code> , but the file does not have to be open.
lstat	Retrieves the same information as <code>fstat</code> , but the file does not have to be open. If the file refers to a symbolic link, <code>lstat</code> returns information about the link, not the file to which the link refers.
getattrlist	Returns a list of attributes for a file system object, such as a volume, directory, file, or file fork. Some information <code>getattrlist</code> returns for a file are its name, its size, the number of forks, and the size of each fork. Mac files have two forks: a data fork and a resource fork.
getdirentries	Returns information about the files and subdirectories inside a given directory.
getdirentriesattr	Returns a list of attributes for multiple directories. It combines the work of <code>getattrlist</code> and <code>getdirentries</code> .
mmap	Maps a file into memory.

Table 10.2 Extra Information `fs_usage` Provides

Information	Prefix	Description
Address	A=	For cache hits, page ins, and page outs, the address tells you where the cache hit, page in, or page out occurred in memory.
Bytes	B=	For file reads and writes, the number of bytes the operating system read from or wrote to the file. For page ins and page outs, the number of bytes that were paged in from disk or paged out to disk. <code>fs_usage</code> reports the number of bytes as a hexadecimal number.
Disk Block Number	D=	For file reads and writes, the physical disk block number being read from or written to.
File Descriptor	F=	An integer that identifies an open file. <code>fs_usage</code> reports a file descriptor for most of the file operations in Table 10.1.
Offset	O=	For <code>lseek</code> operations, the offset tells you how far the operating system moved from the start of the file.
Select Return	S=	The return value of the <code>select</code> system call. A value of 0 means the select timed out.
Error Number	[]	If a file operation produced an error, the error number appears in brackets.

`fs_usage` Options

The `fs_usage` tool provides three options. You can use multiple options in one command. The following command:

```
fs_usage -w -f network AppName
```

Combines the `-w` and `-f` options to limit `fs_usage`'s output to network system calls and tells `fs_usage` to wrap its output to a second line if the window isn't wide enough to display all the output on one line.

-e Option

The `-e` option allows you to exclude certain processes from the `fs_usage` report. Supply a list of process ID numbers or application names.

```
fs_usage -e 449 465 508
```

On Mac OS X the `—e` option does not help much because there are too many processes running. As a test, I started up my Mac, launched the Terminal application, and ran `top`. `top` reported 39 running processes when I had only two applications running: the Finder and Terminal. To look at a few applications, supply the process ID numbers of the applications you’re interested in viewing instead of using the `—e` option with every process you don’t want to see.

-f Option

Normally `fs_usage` displays every system call it finds. The `—f` option filters `fs_usage`’s output. Supply one of five filtering modes: network, file system, cache hit, exec, or pathname. With the network filtering mode,

```
fs_usage -f network AppName
```

`fs_usage` displays only network system calls. With the file system filtering mode,

```
fs_usage -f filesystem AppName
```

`fs_usage` displays only file system calls.

If you want `fs_usage` to report cache hits, you must use the `—f` option and cache hit mode.

```
fs_usage -f cachehit AppName
```

Using the exec filtering mode tells `fs_usage` to show only exec and spawn events.

```
fs_usage -f exec AppName
```

Using the pathname filtering mode tells `fs_usage` to show only pathname-related events.

```
fs_usage -f pathname AppName
```

-w Option

`fs_usage` usually supplies as many columns of data as will fit in your shell window; the wider the window, the more columns of data `fs_usage` shows. When you use the `—w` option, `fs_usage` displays all the available information, wrapping the output to the next line if it won’t fit on one line. The `—w` option saves you from having to resize the window to see all of `fs_usage`’s output.

```
fs_usage -w AppName
```


sc_usage

The `sc_usage` tool maintains a running list of system calls and page faults for a given application. You must use the `sudo` command to run `sc_usage`. Supply either a process ID or the name of a currently running application to `sc_usage`.

```
sudo sc_usage ProcessID
sudo sc_usage AppName
```

When you run `sc_usage` on your application, chances are high no data will appear initially because you're in the Terminal application instead of your application. Switch to your application and do some things in it to see some of the data `sc_usage` records.

What sc_usage Tells You

`sc_usage` provides its output differently than `fs_usage`. `fs_usage` has one listing for each individual event while `sc_usage` has one listing for each type of system call along with the number of calls. If your application has 25 cache hits, `fs_usage` displays 25 `CACHE_HIT` listings while `sc_usage` displays one `cache_hit` listing with a count of 25.

By default `sc_usage` updates its data every second, replacing the old information with the new. This behavior differs from `fs_usage`, which adds each system call to the output immediately. The `sc_usage` output is more useful to view while your program's running rather than saving and viewing later.

`sc_usage` provides its output in three areas. At the top is summary information about the program being monitored. In the center is a list of the system calls the program made along with information about the calls. At the bottom is a list of blocked system calls along with information about each call.

Program Summary Information

At the start of the report is summary information about the program being monitored. `sc_usage` reports eight pieces of summary information.

- The program's name.
- The number of preemptions. A preemption occurs when the operating system interrupts your program to give time to another program. Preemptions are good as a Mac user. They allow you to simultaneously write source code in Xcode, download a file in Safari, and listen to a CD in iTunes.
- The number of page faults. A page fault occurs when the operating system can't find a page of memory in physical memory.

- The number of context switches. A context switch occurs when the operating system switches to another program. A major context switch occurs when you switch applications. If you're in Safari and switch to iTunes, a major context switch changes the foreground process from Safari to iTunes. A minor context switch occurs when the operating system gives time to a program running in the background. If you're surfing the Internet in Safari and listening to music with iTunes, a minor context switch gives iTunes the time it needs to play the music you're listening to.
- The number of system calls your program made.
- The number of threads in the program.
- The current time. It's always important to know what time it is.
- The elapsed time, the amount of time `sc_usage` has been monitoring your program.

When you look at the summary information, remember that the numbers of preemptions, page faults, context switches, and system calls `sc_usage` reports are the numbers that occurred during the last sampling period. If you run your program and switch to `sc_usage`, you could see your preemptions, page faults, context switches, and system calls trickle down to zero. The trickling occurs because `sc_usage`'s default sampling period is one second. If you're examining `sc_usage`'s results, your application won't have much activity in the previous second.

System Call List

After the program summary information is a list of each system call your application makes. `sc_usage` tells you the following information for each system call:

- The system call name.
- The number of times your program made the system call.
- The amount of time your program spent in the system call.
- The amount of time the call has been waiting.

When reporting the number of times your program made a system call and the amount of time the call has been waiting, there may be two values, one of which is in parentheses.

500 (41)

The listing says your program made this system call 500 times since launching `sc_usage`. 41 of the 500 calls occurred during the last sampling period. The first three listings in the system call list are.

- System Idle, which tells you the amount of time the operating system is idle.
- System Busy, which tells you the amount of time the operating system is busy.
- Usermode, which tells you the amount of time spent in your program.

After the System Idle, System Busy, and Usermode listings comes the page fault system calls. Table 10.3 lists these calls.

After the page fault system call listings come the rest of the system call listings. You can see some common system calls in Table 10.1.

Table 10.3 Page Fault Types

Page Fault Type	Description
cache_hit	The operating system found the page of memory in the computer's cache.
page_in	The operating system moved the page from disk to physical memory.
zero_fill	The operating system created the page of memory and filled it with zeroes.
copy_on_write	The operating system copied the memory page from another page of memory.

sc_usage Options

`sc_usage` has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
sudo sc_usage -l -s 10 AppName
```

Combines the `-l` and `-s` options. It tells `sc_usage` to turn off continuous updating of data and to use a sampling interval of ten seconds.

-c Option

The `-c` option lets you specify a code file that contains the system calls you want `sc_usage` to report your program making. You can view the default code file at `/usr/share/misc/trace.codes`. Supply the path to the code file when using the `-c` option.

```
sudo sc_usage -c CodeFile AppName
```

-e Option

The `—e` option sorts the output data by the call count, the number of times your application made the system calls. By default `sc_usage` sorts the listings by the amount of time your program spent in each system call.

```
sudo sc_usage —e AppName
```

`sc_usage` sorts the system calls on a first-come-first-served basis. When your program makes a system call for the first time, the call appears at the bottom of the list. If your program makes multiple system calls for the first time in a given sampling period, `sc_usage` sorts the calls based on the number of times your program made them, and this order never changes. Suppose you have two system calls, A and B. If during the first sampling period, your program calls A 10 times and B 20 times, B appears ahead of A. If your program calls A 100 times the next period and doesn't call B, B still appears ahead of A even though A has been called more than B.

-E Option

Running `sc_usage` with the `—E` option launches your program first. Use the `—E` option when you want to see the system calls your program makes when it starts. Supply a path to your program's executable file and any runtime arguments your program needs to run. For Mac OS X application bundles, the executable file resides three directories inside the application bundle.

```
> Application Bundle
  > Contents
    > MacOS
      > Executable File
```

If you're in the directory where the application bundle resides, you would launch the program with the following command:

```
sudo sc_usage —E AppName.app/Contents/MacOS/AppName
```

-l Option

The `—l` option turns off the continuous updating of data. Every time `sc_usage` updates its data, it appends the output to the shell window. If you use the default sampling rate of one second and watch your program for one minute, you would end up with 60 reports in the shell window. Use the `—l` option if you want a record of all system calls your application makes.

-s Option

The `—s` option lets you specify how often you want `sc_usage` to update its output. Supply the number of seconds you want in the sampling period; the default period is one second. The following command tells `sc_usage` to update its output every five seconds:

```
sudo sc_usage —s 5 AppName
```

vmmap

The `vmmap` tool gives you a map of the virtual memory the operating system reserved for your program. It's the command-line equivalent of the VM Tracker instrument in Instruments. Supply a process ID number or program name when running `vmmap`.

```
vmmap ProcessID  
vmmap AppName
```

What vmmap Tells You

The report `vmmap` generates has three sections.

- The non-writable memory regions, which consist mostly of shared libraries to which you linked your application. Shared libraries being non-writable is a good thing. If you write an application that uses UIKit, you don't want your application to be able to overwrite the memory UIKit is using.
- The writable memory regions.
- A summary report of the virtual memory map.

When you look at the report, keep in mind that it's telling you about the memory the operating system reserved for your application. It does not necessarily mean your application is using all of that memory. When the operating system launches your program, it reserves a large chunk of memory, more than most programs need. From this large chunk of reserved memory, the operating system allocates smaller chunks when your application needs them. The `vmmap` report gives you a map of the initial large chunk of reserved memory. You have to do some digging to discover how much of the reserved memory your program uses.

Non-Writable Memory Regions

The report for the non-writable memory regions your application reserves has one listing for each memory region. `vmmap` tells you the following information for each region:

- The purpose of the memory in the region.
- The starting and ending addresses of the memory region. If you're debugging your program, you can enter the starting address in the debugger's memory browser to view the memory contents.
- The size of the region. It appears in brackets.
- The permissions for the memory region.
- The sharing mode of the memory region.
- Additional data that depends on the memory region. Some regions have no additional data to display. For some regions `vmmap` shows the pathname of the executable file for the memory region. Most of these pathnames will be libraries linked to your program. For some regions `vmmap` shows the contents of the memory in the region. The amount of data that appears in the window depends on the width of your window. Wider windows reveal more memory contents and a larger portion of pathnames.

Region Purpose

The region purpose of a memory region gives you a general idea of the region's contents. The first non-writable memory region for every application has the name `__PAGEZERO` as its purpose. This region is protected; accessing it will crash your program.

Non-writable memory regions have six common purposes.

- `__TEXT` regions contain executable code or constant data, like the constants your application declares.
- `__DATA` regions contain data, which comes as a shock to you. The data is read-only for a non-writable memory region, but a writable memory region would contain data that your application could both read and write.
- `__LINKEDIT` regions contain raw data the dynamic linker uses, such as symbol table entries. These regions generally follow `__TEXT` regions.
- `__OBJC` regions contain data that the Objective-C runtime library uses. Only Objective-C programs have these regions.
- Shared memory regions are shared by multiple applications. System libraries like Cocoa and OpenGL are the major source of shared memory regions.
- Mapped file regions are memory mapped files, where the operating system maps part of a file into a program's virtual address space. Memory mapped files let multiple programs read from and write to the same file.

One not so common purpose deserves some explanation. Guarded memory regions, which show up as `STACK GUARD` or `MALLOC guard page` in a `vmmap` report, prevent out of order accesses. Normally the computer allows operations to be performed out of order. Suppose your program is currently performing a series of integer calculations, followed by some floating-point calculations. If the CPU performed the operations in order, the floating-point unit would be idle until the series of integer calculations finished. To make use of the idle floating-point unit, the CPU performs the floating-point calculations while it performs

the series of integer calculations. The floating-point calculations are out of order operations in this situation. Out of order operations are normally good; they allow your program to run efficiently. Sometimes accessing memory out of order can wreak havoc with your program. Memory that controls I/O devices is especially vulnerable to out of order memory accesses. Making vulnerable memory regions guarded prevents bad things from happening in your program.

Permissions

The permissions for a memory region tell you how you can access the region. `vmmap` lists two sets of permissions for each memory region. The first set lists the permissions for your application, and the second set lists the maximum permissions. The most common use of maximum permissions is running your program in a debugger. In a non-writable memory region the most common permission is.

`r--/rwx`

Which means your application can read memory in this region, but cannot write to memory or execute memory in this region. Your application will never have write permission in non-writable memory regions. The only way a memory region can have execute permission is if the operating system loaded a piece of executable code in the region. An application with maximum permissions can read, write, and execute memory in this memory region. The most common situation to use maximum permissions to write to memory in a non-writable region is a debugger inserting a breakpoint in an application.

If you look at a `vmmap` report, you can see the first listing, `__PAGEZERO`, has six dashes in the permissions area, which means nobody can read, write, or execute memory there. If you've done any work with pointers, you know that accessing null pointers will crash your program. Denying permission to access `__PAGEZERO` ensures a crash if you do anything with a null pointer.

Sharing Modes

The sharing mode of a memory region tells you how the region interacts with other programs. A memory region can have one of seven possible sharing modes.

- Copy on write (COW) regions can be shared at multiple locations in your application or shared by multiple applications. When your application modifies the memory of a copy on write region, it receives a copy of the region, and the page of memory your application modified becomes private. When all of the pages of a memory region become private, the region becomes private as well.
- Private (PRV) regions are visible only to your application.
- Empty (NUL) regions do not exist in physical memory.

- Aliased (ALI) regions point to another memory region.
- Shared (SHM) regions are shared by multiple applications.
- Zero filled (ZER) regions have had their contents cleared and filled with zeroes.
- Shared alias (S/A) regions combine the characteristics of aliased and shared memory regions.

Writable Memory Regions

`vmmap` provides the same types of information for writable memory regions as it does for non-writable ones: purpose, starting address, ending address, size, permission, sharing mode, and possible additional data. When you look at the permissions for a writable memory region, you will see most of the listings give your application write access, which is what you expect for a writable memory region. If a writable memory region shows no additional data, the memory in the region is heap memory or stack memory.

Writable memory regions rarely have the `__TEXT` and `__LINKEDIT` purposes that are common in non-writable regions. Common purposes for writable memory regions include the following:

- `__DATA` regions contain data (surprise) that your application can both read from and write to.
- `MALLOC` regions have been allocated using the function `malloc()`.
- `MALLOC_TINY` regions consist of small memory allocations, allocations 512 bytes and smaller.
- `MALLOC_LARGE` regions consist of large memory allocations. Apple's documentation says large memory allocations are at least 4 pages in size, 16 KB. But I saw `MALLOC_LARGE` regions that were 4 KB when I ran `vmmap`.
- `MALLOC_FREE` regions were allocated earlier, used, and freed when your program was finished with the memory.
- `VM_ALLOCATE` regions are regions that were allocated with the `vm_allocate()` function. This function allocates virtual memory regions. `malloc()` allocates memory from these virtual memory regions.
- `STACK` regions contain stack memory, which is where the operating system places the parameters of every function your application calls.
- `ATS` regions involve the Apple Type Services (ATS) framework, which deals with fonts. If your program works with text, chances are high you'll have ATS memory regions.
- `__OBJC` regions contain data that the Objective-C runtime library uses.
- `__LOCK` regions are locked, which means the operating system can't move them when memory runs low.

You might be wondering where your program calls `malloc()` if you didn't call it in your code. When you use the `new` operator in a C++ program to create an object, you're indirectly calling `malloc()`. When you use Cocoa methods to create things like arrays, strings, and dictionaries, you're calling `malloc()` indirectly.

Summary Report

For non-writable memory regions, `vmmap`'s summary displays the following data:

- The total amount of memory reserved for your application.
- The amount of resident memory, the memory that is in physical RAM.
- The amount of memory that was either swapped out or unallocated. Swapped-out memory is memory that was in RAM, but moved to disk because other programs needed the RAM.

Remember that shared libraries your application links to comprise most of the read-only memory regions. Other applications use these libraries as well so seeing a high amount of resident memory does not necessarily mean your application is a memory hog.

For writable memory regions, `vmmap`'s summary tells you the following information:

- The total amount of memory reserved for your application.
- The amount of memory the computer wrote in your application.
- The amount of resident memory.
- The amount of swapped-out memory. Swapped-out memory can mean one of two things: you're switching to other applications or your program uses a lot of memory.
- The amount of unallocated memory.

The sum of resident memory and swapped-out memory is the best measurement of the amount of memory your program uses.

At the end of the summary report, `vmmap` breaks down the memory map by region purpose and tells you the total amount of virtual memory for each purpose.

vmmap Options

`vmmap` has several options to customize the way it runs. You can combine multiple options in one command. The following command:

```
vmmap -w -submap ProcessID
```

Tells `vmmap` to print wide output and to print submap information in the output.

-d Option

When you use the `—d` option, `vmmap` takes two snapshots of your program's virtual memory map. It takes the first snapshot immediately and takes the second one after a period of time that you specify. After taking the snapshots, `vmmap` reports three pieces of information.

- The memory regions that changed in the second snapshot.
- Memory regions in the first snapshot that aren't in the second.
- Memory regions in the second snapshot that aren't in the first.

The `vmmap` report has one difference listing for each memory region that changed between the first snapshot and the second. The difference listing starts with the text `Diff`, then displays the region as both snapshots recorded it. The usual cause of a difference in a memory region is a change in sharing mode.

After showing the memory regions that changed, `vmmap` lists the memory regions that appear in only one of the snapshots. It starts with the regions that appear only in the first snapshot, then lists the regions that appear only in the second snapshot. Non-writable memory regions appear before writable ones.

Using the `—d` option is relatively simple. Supply the number of seconds you want `vmmap` to wait before taking the second snapshot. The following example waits 30 seconds:

```
vmmap -d 30 AppName
```

-w Option

The `—w` option displays wide output for each memory region. Wide output allows you to see a greater amount of additional data for each region.

```
vmmap -w AppName
```

-resident Option

When you run `vmmap` with the `—resident` option, it lists the virtual and resident size of each memory region. The resident size tells you how much of the region is in physical memory.

```
vmmap -resident AppName
```

-pages Option

When you run `vmmap` with the `—pages` option, it lists the size of each memory region in pages instead of bytes. The default memory page size is 4 KB on Intel Macs, but pages can also be 2 MB and 4 MB.

```
vmmap —pages ProcessID
```

-interleaved Option

When you run `vmmap` with the `—interleaved` option, the report does not separate the memory regions into writable and non-writable regions. It lists the memory regions in order of their starting address, with the lowest memory regions appearing first.

```
vmmap —interleaved ProcessID
```

-submap Option

The `—submap` option tells `vmmap` to print information about memory submaps. A *submap* is an area of memory the operating system can use in multiple applications. A Cocoa runtime library may reside in a submap so any running Cocoa application can use it.

In a `vmmap` report all listings below a submap listing belong to that particular submap until another submap listing appears.

```
vmmap —submap AppName
```

-allSplitLibs Option

The `—allSplitLibs` option tells `vmmap` to print information about all shared system split libraries. The default behavior is to print information about only the libraries your program loads. *Shared system split libraries* are dynamic libraries that multiple programs share. Examples of these libraries are the Cocoa and OpenGL libraries. The `—allSplitLibs` option generates a lot of unused split lib listings for non-writable regions.

```
vmmap —allSplitLibs AppName
```

-noCoalesce Option

The `-noCoalesce` option tells `vmmap` not to coalesce adjacent identical memory regions. When you use this option, `vmmap` generates more listings for writable and non-writable memory regions.

```
vmmap -noCoalesce AppName
```

-v Option

The `-v` option tells `vmmap` to generate verbose output. The `-v` option combines the `-w`, `-resident`, `-submap`, `-allSplitLibs`, and `-noCoalesce` options.

```
vmmap -v AppName
```

heap

The `heap` tool lists the objects allocated in your application's heap. Supply a process ID or an application name when running `heap`.

```
heap ProcessID  
heap AppName
```

heap's Output

The output `heap` generates comes in three levels. First, `heap` reports a summary for each memory zone. The summary tells you the following information for each zone:

- The size of the zone.
- The number of nodes allocated.
- The amount of memory allocated.
- The largest unused block of memory in the zone.

After reporting each memory zone, `heap`'s summary reports the total number of nodes allocated with the total amount of allocated memory. What does it mean when a node is allocated? It means your program made a memory allocation. One node equals one memory allocation.

Second, **heap** reports the size of each memory allocation. For each memory zone **heap** tells you the number of nodes allocated along with the size listings. In each size listing, **heap** reports the size of the memory allocation with the number of allocations in brackets. The following listing:

```
24KB[ 7 ]
```

Says your program made seven memory allocations of 24 KB in the memory zone.

heap sorts the size listings by allocation size, with the largest allocations coming first. After listing the information for each zone, **heap** reports the same information for all zones combined.

Finally, **heap** reports the objects your program allocated. The report starts with the total number of Objective-C classes and Core Foundation types allocated on the heap for the whole application. Then for each memory zone, it lists the objects that were allocated in that zone. For each class **heap** tells you the following information:

- The number of objects of the class your program allocated.
- The amount of memory allocated.
- The average amount of memory allocated, which is the total amount of memory allocated divided by the number of objects.
- The class name. Memory not part of a class has the class name **non-object**.
- The type of class. Common class types are ObjC, C++, and CFType.
- The binary, which in most cases is the framework or library the class is part of.

heap sorts the listings by the number of allocations, with the highest numbers appearing first in the list.

heap Options

heap has several options to customize the output it displays. You can combine multiple options when running **heap**. The following command tells **heap** to use both the **-guessNonObjects** and **-showSizes** options:

```
heap -guessNonObjects -showSizes AppName
```

-guessNonObjects Option

The `-guessNonObjects` option tells `heap` to search the memory of each object for pointers to non-objects, blocks of memory allocated by `malloc()`. The `heap` output has one listing for each of these memory blocks. The block listings show the Objective-C object, with the offset from the start of the object in brackets. The following listing:

```
NSCFArray[ 36 ]
```

References a block of memory that is located 36 entries from the start of the array.

-sumObjectFields Option

Like the `-guessNonObjects` option, the `-sumObjectFields` option tells `heap` to search the memory of each object for pointers to non-objects. When `heap` finds a pointer to a non-object, `heap` adds the size of the non-object to the object's listing.

-showSizes Option

When `heap` lists the objects in a memory zone, it has one listing for each class. The listing for a class contains all the memory allocations of that class. If you use the `-showSizes` option, `heap` creates a listing for each memory allocation size in every class.

Suppose your Cocoa application allocates strings of length 32, 64, 96, and 128. `heap` normally combines all the string allocations into one listing for a memory zone. If you pass the `-showSizes` option to `heap`, it prints separate listings for the 32, 64, 96, and 128-byte string allocations.

-addresses Option

When you supply the `-addresses` option to `heap`, it prints the address of each object on the heap. Tell `heap` what objects to print. Supplying `all` tells `heap` to print the address of each object on the heap.

```
heap -addresses all AppName
```

If you don't want to print the address of each object on the heap, you can supply a regular expression that tells `heap` what objects to print. The following command prints the address of every `NSCFDictionary` object found on the heap:

```
heap -addresses NSCFDictionary AppName
```

If your regular expression consists of something more than a single class name, place the expression in single quotes. The following command prints the address of every Cocoa object found on the heap:

```
heap -addresses 'NS.*' AppName
```

leaks

The `leaks` program checks your application for memory leaks, which you can also do in Instruments with the Leaks instrument. `leaks` is easier to use for command-line applications.

Running leaks

You can run `leaks` without switching to `root`. Supply either a process ID or application name.

```
leaks AppName
leaks ProcessID
```

Setting the environment variable `MallocStackLogging` to 1 turns on call stack recording, which tells `leaks` to display the call stack for each memory leak. Turning on call stack recording is a good idea. If you have a memory leak in your program, the first thing you want to know is where you're leaking memory so you can fix the leak in your code. By turning on call stack recording, `leaks` can tell you where you're leaking memory. Refer to the section "Setting Environment Variables for Debugging" in Chapter 7, "Debugging", for instructions on setting environment variables.

What leaks Tells You

`leaks` provides two pieces of information about your program. The first piece of information is a summary that reports the following information:

- The number of nodes your application allocated, which is the number of memory allocations your program made.
- The amount of memory your program allocated.
- The number of memory leaks.
- The total amount of leaked memory.

The second piece of information is a listing for each memory leak that `leaks` found. Each listing reports the following information:

- The address of the leaked memory.
- The size of the leak.
- A memory dump of the leak. For memory leaks 128 bytes and smaller, `leaks` shows the contents of the leaked memory. For memory leaks larger than 128 bytes, `leaks` displays the first 128 bytes of leaked memory.
- If the leak occurs in a Core Foundation or Cocoa class, `leaks` reports the name of the class.
- If you set the `MallocStackLogging` environment variable to 1, `leaks` displays the call stack of functions leading to the memory leak.

leaks Options

`leaks` has three options you can supply. You can use multiple options. The following command:

```
leaks -nocontext -nostacks
```

Tells `leaks` to conceal the memory dump and call stack in the individual memory leak listings.

-nocontext Option

The `-nocontext` option tells `leaks` to suppress the memory dump. Use the `-nocontext` option if you don't care about the contents of the memory your program is leaking.

```
leaks -nocontext AppName
```

-nostacks Option

Setting the environment variable `MallocStackLogging` to 1 tells `leaks` to display the call stack for each leak. The `-nostacks` option tells `leaks` to suppress the call stack display. I'm not sure why you would want to suppress the call stack display. The call stack display helps you discover where the memory leaks are in your code.

```
leaks -nostacks ProcessID
```


-exclude Option

When you use the `—exclude` option, `leaks` reports the summary information: the amount of memory allocated, the number of memory leaks, and the total amount of leaked memory. If the function you supply appears in the call stack of a memory leak, `leaks` does not create a listing for that memory leak. Use the `—exclude` option to keep functions you already know leak memory and functions falsely accused of leaking memory from creating unnecessary memory leak listings.

```
leaks —exclude FunctionName ProcessID
```

malloc_history

As its name suggests, `malloc_history` supplies a history of every memory allocation your application makes using the `malloc` library. The `Allocations` instrument in Instruments provides similar information, but `malloc_history` works better with command-line programs and makes saving the results to a text file easier.

To run `malloc_history` on your application, you must turn on the `malloc` library's debugging capabilities by setting the environment variable `MallocStackLogging` to 1. While you're setting environment variables, you may also want to set the variables `MallocStackLoggingNoCompact` and `MallocScribble` to 1. Setting the `MallocStackLoggingNoCompact` environment variable allows you to run `malloc_history` on a specific memory address. Setting the `MallocScribble` environment variable causes the operating system to overwrite memory your application frees. Overwriting freed memory helps you detect memory smashers, places in your code that overwrite memory. Refer to the section "Setting Environment Variables for Debugging" in Chapter 7, "Debugging", for instructions on setting environment variables.

Running malloc_history

There are two ways to run `malloc_history` on an application.

```
malloc_history AppName —all_by_size
malloc_history AppName —all_by_count
```

`malloc_history` displays the memory allocations your application made in the shell window with one listing for each combination of allocation size and call stack. A listing tells you the following information:

- The thread.
- The number of memory allocations.
- The amount of memory allocated.
- The call stack for the allocation.

The only difference between `-all_by_size` and `-all_by_count` is how `malloc_history` sorts the listings. `-all_by_size` sorts by allocation size, with the largest sizes appearing first while `-all_by_count` sorts by allocation count, with the largest counts appearing first.

Running `malloc_history` on a Specific Memory Area

`malloc_history` normally lists every memory allocation your program makes. You can tell `malloc_history` to report allocations made in a particular memory area by supplying a memory address to `malloc_history`.

```
malloc_history ProcessID MemoryAddress
```

`malloc_history` provides a history of all memory allocations that manipulated memory at the address you specified. For each memory allocation, `malloc_history` reports the size of the allocation (It says `arg =`), the thread, and the call stack.

To run `malloc_history` on a memory address instead of an entire program, you must set the environment variable `MallocStackLoggingNoCompact` to 1.

Showing All Allocation Events

If you run `malloc_history` with the `-all_events` option,

```
malloc_history ProcessID -all_events
```

`malloc_history` lists every memory allocation and free event. Using the `-all_events` option gives you much more output than the `-all_by_size` and `-all_by_count` options.

Chapter 11

OpenGL Tools

One of Apple's goals with Mac OS X was to make it the premier OpenGL platform for both users and developers. To make Mac OS X the best OpenGL platform for developers, Apple includes OpenGL developer tools as part of the Xcode Tools. This chapter explains how to use these tools.

If you are running Xcode 4.3 or later, the Mac OpenGL tools I cover in this chapter are not packaged with Xcode. Choose Xcode > Open Developer Tool > More Developer Tools to go to Apple's developer downloads page. Download the Graphics Tools for Xcode package to install the OpenGL tools. Choose Xcode > Open Developer Tool > OpenGL ES Performance Detective to launch OpenGL ES Performance Detective. In Xcode 4.5 and later OpenGL ES Performance Detective is integrated into Xcode's frame debugger.

OpenGL Profiler

OpenGL programs demand high performance to be usable, making these programs the most likely to require profiling. But if you profile an OpenGL program with the Time Profiler instrument in Instruments, you'll notice a problem. The call tree doesn't tell you much information about the OpenGL function calls the program makes. Use OpenGL Profiler to profile your program's OpenGL function calls.

OpenGL Profiler works like a normal profiler, but it focuses exclusively on OpenGL function calls. For each OpenGL function call your program makes, OpenGL Profiler reports the number of times your program called the function and the amount of time spent in the function. In addition to profiling, OpenGL Profiler helps you debug your OpenGL programs. You can set breakpoints, run scripts of OpenGL commands, and view the contents of OpenGL's buffers.

Only Mac applications can use OpenGL Profiler. iOS applications should use the OpenGL ES Analyzer instrument in Instruments.

Choosing a Program to Profile

When you launch OpenGL Profiler, the main window (see Figure 11.1) opens. Your first step is choosing a program to profile. How you choose a program to profile depends on whether or not the program you want to profile is currently running. If the program is running, select the Attach to application radio button. A list of all running applications appears in the window. Select the program you want to profile from the list.

If the program you want to profile isn't running, select the Launch application radio button. A list of the programs you previously profiled appears in the window. If the program you want to profile isn't on the list, click the + button. An Open panel opens for you to choose the program to profile. If your program takes any runtime arguments, enter them in the Launch Arguments column.

If you choose to launch your application from OpenGL Profiler, the Launch Settings disclosure triangle becomes enabled. Click the disclosure triangle if you want to use a custom pixel format, launch your application in 64-bit mode, or set environment variables. You must build a 64-bit version of your application to be able to launch it in 64-bit mode.

Custom Pixel Formats

Looking at the Use custom pixel format checkbox in Figure 11.1, you might be wondering what a custom pixel format is. Pixel formats allow you to specify attributes for the draw context where OpenGL does its drawing. Do you want your program to run in a window or run fullscreen? Do you want a back buffer? Do you want your drawing hardware-accelerated? What color depth do you want: 16-bit, 32-bit, or 64-bit color? How large do you want OpenGL's alpha, depth, stencil, and accumulation buffers? Pixel formats answer these questions. You have to specify a pixel format in your code for your OpenGL program to run. Using a custom pixel format lets you experiment with pixel format attributes without having to recompile your program.

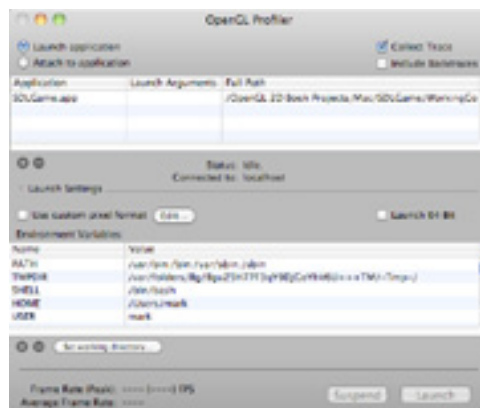


Figure 11.1

OpenGL Profiler
main window

To build your custom pixel format, click the Edit button. The custom pixel format window opens, which you can see in Figure 11.2. Click the Attributes button to open the drawer containing the pixel format attributes you can set. Double-click an attribute to add it to the custom pixel format. To add multiple attributes, select them from the drawer and drag them to the window. Double-click the Value column to change an attribute's value. For Boolean values, use `GL_TRUE` or 1 for true and `GL_FALSE` or 0 for false. To remove an attribute from the custom pixel format, select the attribute from the custom pixel format window and press the Delete key. Clicking the Clear button removes all the attributes from the pixel format. Close the window when you're finished. Select the Use custom pixel format checkbox to tell OpenGL Profiler to use the custom pixel format you built.

Setting Environment Variables

To add an environment variable, click the + button below the environment variable list. Enter the variable's name in the Name column and the variable's value in the Value column.

Remote Profiling

Remote profiling allows you to run your program on one computer and run OpenGL Profiler on another. Running your program on one computer and OpenGL Profiler on another helps tremendously if your program runs fullscreen. Profiling a fullscreen program on a Mac with one monitor is difficult because you can't see the OpenGL Profiler windows when your program runs. Using two Macs lets you see the OpenGL Profiler windows without having to pause your fullscreen program.

To turn on remote profiling, open OpenGL Profiler's preferences. Enter a password for your Mac in the Remote profiling password text field to enable the Enable remote profiling checkbox. Select the checkbox to turn on remote profiling.

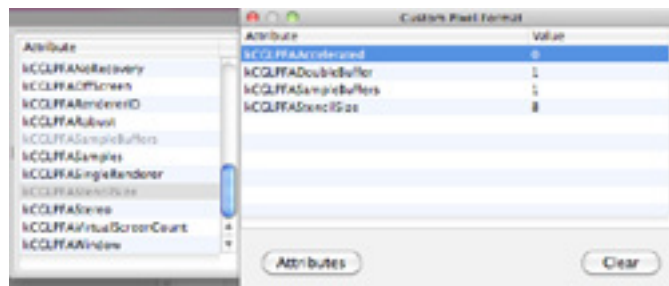


Figure 11.2

Custom pixel
format window

Setting Breakpoints

To be able to perform certain tasks, such as viewing the contents of OpenGL buffers and resources, you have to set breakpoints in OpenGL Profiler. Breakpoints in OpenGL Profiler work similarly to breakpoints in Xcode's debugger. They pause your program. The main difference is in OpenGL Profiler you set a breakpoint when your program calls an OpenGL function, not when it reaches a particular line of code.

Choose Views > Breakpoints to open the breakpoints window, which you can see in Figure 11.3. You can set a breakpoint at any OpenGL function call and at any Core OpenGL (CGL) function call. CGL is the way Mac OS X programs access OpenGL. If you use the Cocoa OpenGL classes, you're indirectly using CGL; the Cocoa classes sit on top of CGL.

On the left side of the window, you'll see an alphabetical list of functions with three columns: Before, After, and Execute. All the functions initially have the Execute column selected, which means the functions execute normally. Disabling the Execute column for a function means OpenGL Profiler skips the function when it finds a call to that function in your program. Core OpenGL functions must execute so OpenGL Profiler prohibits you from disabling Execute for those functions.

You can set a breakpoint before an OpenGL function call, after, or both. Setting a breakpoint before and after a function lets you see the function's effects on OpenGL's buffers. Click the Before or After column to set a breakpoint on a function. A blue dot signals that you set a breakpoint.

In addition to setting breakpoints for individual functions, you can tell OpenGL Profiler to pause execution when your program commits an OpenGL error. Below the function list is a series of checkboxes to break on various types of OpenGL errors. You can break on the following errors as well as temporarily ignore breakpoints:

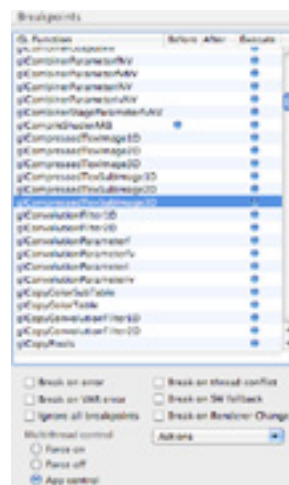


Figure 11.3

Breakpoints window

- Any OpenGL error.
- A VAR error, which occurs when there is a problem using the vertex array range extension.
- A thread conflict.
- SW fallback, which occurs when a graphics card does not support a particular feature and OpenGL has to fall back to its software implementation.
- Renderer change.

Multithread Control

The OpenGL framework can do its work in a separate thread, which can improve the performance of OpenGL applications on Macs with multiple CPU cores. The Multithread control radio buttons let you control OpenGL's multithreading. You can turn multithreading on and off (turning it off makes debugging easier), or you can choose to control it from your application. To turn on multithreading from your application, call the function `CGLEnable()` and supply the constant `kCGLCEMPEngine` as the second argument.

Breakpoint Actions

Below the checkboxes is the Actions menu. Use the Actions menu to set breakpoints for all functions, break before or after draw calls, and attach scripts to breakpoints. Refer to the section "Scripts Window" later in this chapter for more information on attaching scripts.

Profiling Your Program

Before you start to profile, you must tell OpenGL Profiler what to record as your program runs. There are two checkboxes in the upper right corner of the main window. Selecting the Collect Trace checkbox tells OpenGL Profiler to record a list of all the OpenGL function calls your program makes as it runs.

Selecting the Include Backtraces checkbox tells OpenGL Profiler to record the call stack. Recording the call stack lets you see what functions in your program made the OpenGL calls. Your application will run slower in OpenGL Profiler if you record the call stack.

Click the Launch button to launch your program. The program continues to run until it hits a breakpoint or you click the Suspend button. Click the Resume button to continue profiling. Clicking the Force Quit button quits the program you're profiling.

Viewing the Profiling Data

The main OpenGL Profiler window doesn't provide much profiling data about your program. It tells you the current and peak frame rates. To see more data about your program's performance, you must use OpenGL Profiler's auxiliary windows, which you can open from the Views menu.

Trace Window

The trace window shows every OpenGL function call your program made. The window lists the function calls in the order your program made them, one function call per line. By examining the trace you can find unnecessary function calls, improving your program's performance. Looking at the function trace also helps you debug your program. You can learn your program is passing bad data to OpenGL functions and discover missing functions, functions your program should be calling but isn't calling.

The trace window has controls to show more data in the trace window. Selecting the Line #s checkbox adds a line number to each listing. Selecting the Context checkbox adds the OpenGL context to each listing. Selecting the Timing checkbox adds the amount of time your program spent in the function. The trace window reports the time in microseconds. Selecting the Verbose checkbox adds the amount of time your program spent in each function along with two timestamps. The first timestamp is when your program entered the function, and the second timestamp is when your program exited the function. Selecting the RenderID checkbox adds the renderer's memory address to the trace window.

Clicking the Call Stack button opens the call stack drawer. Clicking a function's link from the trace window displays the function's call stack in the call stack drawer. The Include Backtraces checkbox must be selected in the OpenGL Profiler window for function names to appear in the call stack drawer. If you don't include backtraces, the call stack drawer shows only the memory address of each function.

Clicking the Save As Text button saves the trace data to a text file. At the bottom of the Save panel is a pop-up menu that lets you control what trace data is saved. The initial value is Save All, which saves the entire trace to a text file. Choosing Save Selected saves the parts of the trace you selected to a text file. Make sure you select some entries in the trace before you save or else you'll create an empty file. Choosing Save Current View saves the trace entries that are visible in the trace window.

Statistics Window

The statistics window, shown in Figure 11.4, shows the profiling statistics of the OpenGL functions your program calls. The bottom of the window shows the total amount of time, measured in microseconds, your program spent in OpenGL function calls along with the percentage of time your program spent in OpenGL. For each OpenGL function your program calls, the statistics window tells you the following information:

- The number of times your program called the function.
- The total time, measured in microseconds, your program spent in the function.
- The average time, measured in microseconds, your program spent in the function.
- The percentage of the OpenGL time your program spent in the function. The percentages should add up to 100%.
- The percentage of time your program spent in the function.

Resources Window

The resources window lets you examine your application's textures, programs, shaders, FBOs (Frame Buffer Objects), renderbuffers, VBOs (Vertex Buffer Objects), and VAOs (Vertex Array Objects). You must set breakpoints if you want to examine your OpenGL program's resources. The resources are visible only when OpenGL Profiler stops at a breakpoint.

There are tabs at the top of the window to look at a particular resource category. When you choose a category, say textures, that category's instances fill the left side of the resources window. Select an instance to examine it.

For FBOs, VBOs, and VAOs, OpenGL Profiler shows a list of attached objects.

Textures and Renderbuffers

Textures and renderbuffers are both images so OpenGL Profiler displays similar information for each. Select a texture or renderbuffer from the list running along the left side of the resources window. In the center of the resources window is the image itself. Above the texture or renderbuffer is information about it like its size, target (1, 2, or 3-dimensional image), and format. Below the image are options for displaying the texture or renderbuffer in the resources window. For textures click the Viewer Settings button to see the options. These options include the following:

- Zoom level, which lets you get a closer look at the image.
- Source blend mode.
- Destination blend mode.
- Background color.
- Background opacity.

The blend modes determine how the incoming fragment's color blends with the texture environment's color to create a final color for the pixel.

For textures you have the added option of specifying a mipmap level. Mipmaps are smaller versions of a texture used to provide level of detail. When the object moves farther away from the camera, OpenGL uses a smaller version of the texture. Cut the size of the texture in half until you reach a 1-by-1 pixel mipmap. If the base texture is 64-by-64 pixels, the mipmaps would be 32-by-32, 16-by-16, 8-by-8, 4-by-4, 2-by-2, and 1-by-1. With OpenGL Profiler you can view these smaller textures.

Programs and Shaders

Programs and shaders are programs that are meant to run on the graphics card instead of the CPU. The difference between programs and shaders is the language you use to write them. A program is lower level, using a syntax that resembles assembly language. Shaders use GLSL, which has a syntax similar to C.

You can view the source and a log for each shader. You can modify the shader inside OpenGL Profiler and compile shaders as well.

Scripts Window

Use the scripts window, shown in Figure 11.5, to write scripts that execute when your program reaches a breakpoint. You can include any OpenGL function call in a script, but a script can contain only OpenGL function calls.

Click the + button to create a script. OpenGL Profiler adds it to the script list. Double-click the name to change the script's name. Use the text editor in the top half of the window to create the script.

There are two ways to run your script: manually and automatically. To run your script manually, your OpenGL application must be stopped at a breakpoint. Open the scripts window, select your script, and click the Execute button to run the script. If your script has any syntax errors, they appear in the log, which is below the text editor.

To run your script automatically when you reach a breakpoint,

1. Open the breakpoints window by choosing Views > Breakpoints.
2. Select a function from the list.
3. Choose Attach Script from the Actions menu.
4. A sheet opens with a list of scripts to attach. Choose a script from the list.
5. Decide when you want the script to run: before or after reaching the function.
6. Decide if you want your program to continue after executing the script.
7. Click the Attach button.

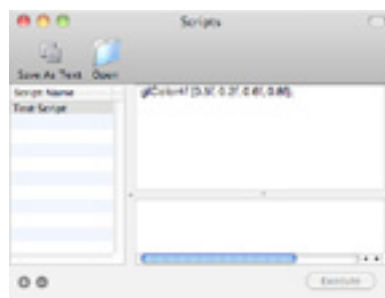


Figure 11.5

Scripts window

To detach a script from a function, select the function from the function list and choose Remove Script from the Actions menu.

Breakpoints Window

I covered the setting of breakpoints earlier, but there are two pieces of information that appear in the breakpoints window when your program reaches a breakpoint. The first piece of information is the call stack, the list of functions your program called leading up to the OpenGL function where you set the breakpoint. The second piece is a snapshot of all the OpenGL state variables when you hit the breakpoint. The call stack and OpenGL state variables help tremendously during debugging. There are tabs to toggle between viewing the call stack and OpenGL state variables, even though they don't appear in the breakpoints window shown in Figure 11.3.

When viewing OpenGL state variables, there are checkboxes to show the changes that occurred since the last breakpoint and the changes from OpenGL's default state. The states that changed since the last breakpoint appear in red text, and the states that changed from the default state appear in blue text. Magenta text means the OpenGL state both changed since the last breakpoint and changed from the default state. Click the Save As Text button to save the state information to a text file.

Pixel Format Window

The pixel format window displays the pixel format information for each OpenGL context your application uses. The information the pixel format window displays is the same information you can set when using custom pixel formats.

Messages Window

The messages window displays the log messages OpenGL Profiler generates as your program runs. These messages can help when debugging your application.

OpenGL Driver Monitor

OpenGL Driver Monitor collects statistics about your graphics card and its interaction with the CPU. It works at a lower level than OpenGL Profiler and works with the graphics card instead of a single application. Use OpenGL Driver Monitor to learn about things like how much video memory you're using, how much time the CPU spent waiting for the graphics card, the number of times the card swapped buffers, and the number of textures loaded on the card.

Getting Started

Launching OpenGL Driver Monitor should open a monitor window for your Mac's graphics card. If no window opens, choose **Monitors > Driver Monitors > DriverName** to open the monitor window, which you can see in Figure 11.6. Initially there's nothing to look at because you haven't given OpenGL Driver Monitor any parameters to monitor. Click the **Parameters** button to open the parameters drawer. Double-click a parameter to add it to the watch list, which contains the parameters whose statistics you want to collect. You can also select parameters from the drawer and drag them to the parameter watch list at the bottom of the monitor window.

Now that you've added parameters to watch, run your OpenGL program. Lines should start appearing on the graph, which means OpenGL Driver Monitor is collecting statistics. If no lines appear, make sure the button below the graph says **Pause** and not **Continue**. If it says **Continue**, click the button. OpenGL Driver Monitor's default sampling rate is one second, which means it plots a point on the graph every second. You can change the sampling rate in the preferences window.

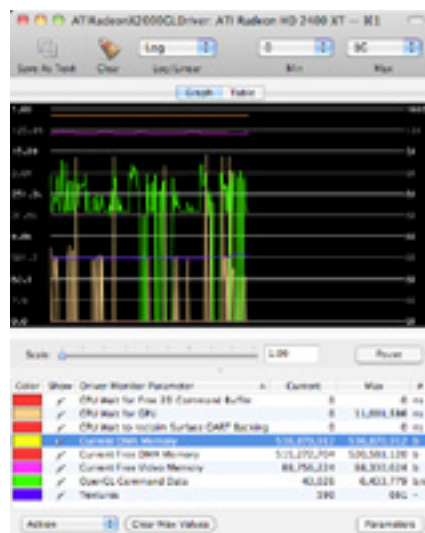


Figure 11.6

Monitor window

Customizing the Graph

The parameter watch list is where you change a parameter's color on the graph. The list has six columns.

- Color, which is the parameter's color on the graph.
- Show. If the Show column has a check mark, the parameter appears in the graph.
- Driver Monitor Parameter, the name of the parameter.
- Current, the current value of the parameter.
- Max, the highest value OpenGL Driver Monitor has sampled.
- #, the unit of measurement. Common units are b for bytes and ns for nanoseconds.

To change a parameter's color, select it from the watch list. Double-click the Color column. A window opens for you to set the parameter's color. Give each parameter its own color to make the graph easier to read.

There are pop-up menus at the top of the window that let you set the minimum and maximum values the graph displays. The Linear/Log button controls the scale of the graph, the values running along the left side of the graph. The linear scale has even scales. The logarithmic scale uses uneven scales to fit as much data in the graph as possible. The data determines the scales in the logarithmic scale while the minimum and maximum values determine the scales in the linear scale. Look at Figure 11.7 to see the difference between the two scales.

Table View

If you don't want to fiddle with graph colors and just want to see the raw data, use the table view. Click the Table tab to switch from the graph to the table view. The table view has one column for each parameter in the watch list. After each sampling period, one second by default, OpenGL Driver Monitor adds a row to the table that contains the newly sampled data.



Figure 11.7

Linear (left) and logarithmic scale for the values 44, 76, 497216, 131, and 8671. Notice how the logarithmic scale prevents the value 497216 from obscuring the differences in the other values.

Renderer Info

Choosing Monitors > Renderer Info opens the renderer window, which displays information about the OpenGL renderers on your Mac. Every Mac has at least two renderers: the renderer for the graphics card and the software renderer. OpenGL Driver Monitor reports lots of information for a renderer, including the following:

- The amount of video memory.
- The amount of texture memory.
- The OpenGL extensions it supports.
- The available sizes for the depth and stencil buffers.
- The available color depths for the color and accumulation buffers.

OpenGL Shader Builder

OpenGL Shader Builder lets you create, test, and debug shaders. Shaders are programs that are meant to run on the graphics card instead of the CPU. The shader replaces a portion of the fixed-function OpenGL pipeline, giving you more control and flexibility in drawing a scene.

OpenGL has three types of shaders: vertex shaders, fragment shaders, and geometry shaders. A vertex shader gives you control over the transformation (converting a scene from world space to screen space) and lighting stage of the 3D graphics pipeline. Instead of feeding vertices to OpenGL and letting it take care of the transformation and lighting, each vertex goes through the vertex shader. Tasks you can perform in a vertex shader include the following:

- Vertex transformation, weighting, and blending
- Normal transformation, rescaling, and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

A vertex shader doesn't have to do all the tasks I listed, but if you don't perform one of the tasks in the vertex shader, OpenGL won't perform the task for you. If your vertex shader generates texture coordinates, OpenGL won't transform the texture coordinates for you. You'll have to transform them yourself.

Fragment shaders, sometimes referred to as pixel shaders, operate on fragments. You can think of a fragment as a pixel wannabe. The fragment contains data for a pixel, such as its primary color and position. OpenGL performs a series of tests on each fragment to determine

whether or not the fragment becomes a pixel. Fragment shaders generate a final color and a depth value for the fragment, from which OpenGL performs its tests. Fragment shaders can perform the following tasks:

- Texture access.
- Texture application.
- Color sum, which creates a final color from the fragment's primary and secondary colors.
- Fog, blending a fog color with the fragment's color.
- Operations on interpolated values in the texture, color sum, and fog stages.

A fragment shader doesn't have to do all the tasks I listed, but OpenGL won't perform the task for you if the fragment shader doesn't do it. If your fragment shader performs fog operations, OpenGL won't perform color summing for you. You'll have to do any color summing yourself.

A geometry shader generates geometry. It takes graphics primitives, such as points, lines, and triangles, as input and creates new primitives as output. Geometry shaders can be used to create procedural geometry, add detail to meshes, and break down complex polygons into groups of simpler polygons.

Creating a Project

To use OpenGL Shader Builder, you must create a project. A project contains shader source files and textures. A project can contain as many shaders as you want, but you might find testing individual shaders difficult if you have dozens of shaders in a project.

When you launch OpenGL Shader Builder, it creates an empty project for you. To create additional projects, choose File > New > Project. A window opens, asking if you want to create shader templates. Creating a shader template adds a shader to the project. Select the checkboxes of the templates you want to create and click the Create button. If you want to create an empty project, click the No Templates button.

The project window initially shows the program view, which you can see in Figure 11.8. The program view has three sections. The top section has a button to add existing shaders to your project and a checkbox to automatically link your project. If your project has a geometry shader, there are controls that let you specify the types of input and output to the shader along with the number of vertices for the shader's output.

The center section contains a list of the shaders in your project. For each shader, there are three columns of information.

- A checkbox indicating whether or not the shader is active.
- The shader's name.
- The type of shader.

The bottom section reports any linking and validation errors in your project. The link log shows any linking errors. The validation log shows any validation errors in your project's shaders. The link results show what the linker generated.

Adding Shaders

To add a blank shader to your project, choose File > New. There are five shaders you can create.

- GLSL vertex shader
- GLSL fragment shader
- GLSL geometry shader
- ARB vertex program
- ARB fragment program

When you add a new shader, it initially has the name Untitled. Save the shader to name it.

If you have shaders you've written that you want to use in the project, click the Add Shaders button to add them. The Add Shaders button is at the top of the project window. Click the Program tab if you don't see the button in your project window.

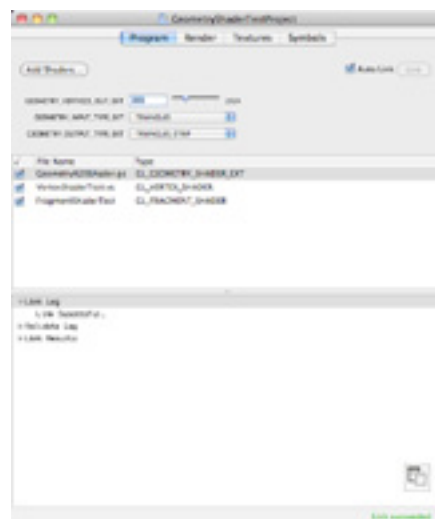


Figure 11.8

OpenGL Shader
Builder project
window

There are two languages you can use to write shaders: GLSL and ARB. GLSL has a C-like syntax, and ARB has an assembly language syntax. GLSL is better for writing large shaders. ARB's advantage is it's older, which means more Macs support it. Unless you need to support versions of Mac OS X earlier than 10.4.3 or support old graphics hardware, use GLSL. Every Intel Mac supports GLSL. Every iPhone and iPad with OpenGL ES 2 support can use GLSL.

A geometry shader requires a vertex shader to accompany it. If you add a geometry shader to a project, make sure the project also has a vertex shader. Geometry shaders let you specify the input type, the output type, and the number of vertices the shader outputs.

Writing a Shader

Double-click a shader in the shaders list to open it in an external editor. Use the editor to write the shader. At the bottom of the editor is a compile log that reports any compiler errors in your shader.

Adding Textures

Each project comes with a texture that can be applied to models for testing purposes. That texture's image is OpenGL Shader Builder's application icon. If you want to use your own images as textures in OpenGL Shader Builder, click the Textures tab in the project window. A list of texture units runs along the left side of the window with an image well for each unit. Drag the image to the image well of the texture unit you want to use. If your shader uses texture unit 2, drag the image to texture unit 2.

For your images to appear on the models in OpenGL Shader Builder, your shader must use the texture unit where you dragged the image. If you drag an image to texture unit 3, but your shader doesn't do anything with texture unit 3, your image is not going to appear on any of the OpenGL Shader Builder models.

Looking at Variables

Click the Symbols tab to view the variables in your project. You can examine and modify uniform variables in GLSL. Uniform variables are variables an OpenGL program sends to a shader, and they are read-only in a GLSL shader. OpenGL Shader Builder lets you experiment with the values of uniform variables.

What you can set for a GLSL uniform variable depends on the variable's type. A simple variable like `sampler2D` has one component: `x`. A 4-by-4 matrix has 16 components: 4 rows multiplied by 4 columns.

For ARB vertex and fragment programs you can examine and modify environment and local parameters. Each parameter represents a color and has four components.

- x is the red component.
- y is the green component.
- z is the blue component.
- w is the alpha component.

Each component in a GLSL or ARB shader has three text fields, which you can see in Figure 11.9. Use the left text field to set the minimum value, which is initially 0. Use the right text field to set the maximum value, which is initially 1. Acceptable minimum and maximum values depend on the component. Use the center text field to specify the current value of the component. Dragging the slider also changes the current value.

If you select the Animate checkbox, the current value of the component constantly changes. Use the text field next to the checkbox to specify how much the value should change. The initial change value is .01. If you keep the initial values, the current value will change from .01, .02, .03, and so on all the way to 1, then go down to .99, .98, .97, and so on all the way back to 0. After selecting the Animate checkbox you may want to click the Render tab to see the effects of changing the component's value.

Compiling a Project

OpenGL Shader Builder is initially set up to automatically compile your shaders. At the bottom of each shader window is the compile log. It will report any errors in your shader. The error reporting is constant, which means errors will appear in the compile log as you're typing your code. The errors will go away when you finish the line of code, assuming there are no errors in the line you finished.

If you don't like the automatic compilation, deselect the Auto Compile checkbox at the top of the shader window. Click the Compile button to compile the shader.

OpenGL Shader Builder is also initially set to automatically link your project. If you don't want to automatically link your project, deselect the Auto Link checkbox in the program view. Click the Link button to link the project.

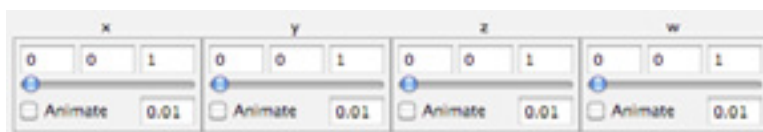


Figure 11.9

Component text fields

Testing a Shader

After writing a shader, you should test it to make sure it works. OpenGL Shader Builder simplifies shader testing. Click the Render tab in the project window to see the shader applied to a model. There is a pop-up menu beneath the view to choose a model.

Click on the model, hold down the mouse button, and move the mouse to rotate the model. Click the Reset View button to set the model back to its initial state.

Benchmarking

Benchmarking measures how fast the project runs. To benchmark your project, click the Render tab in the project window. Click the Benchmark button to open the benchmark window. Enter the amount of time you want to benchmark in the Run Time text field. Click the Run button. After the amount of time you entered passes, the benchmark window tells you the frames per second.

Window Layouts

Initially OpenGL Shader Builder has one window for the project with four tabs: Program, Render, Textures, and Symbols. You might want separate windows open, such as having the render window open so you can see the changes you make to a shader.

In the lower right corner of the project window is the Detach button. Clicking the Detach button creates a separate window for the currently selected tab. Closing the separate window reattaches it to the main project window. The render, textures, and symbols windows have a Reattach button in the lower right corner. Clicking the Reattach button reattaches the window to the main project window.

Using Your Shaders in an OpenGL Program

You've written your shaders, tested them in OpenGL Shader Builder, and they ran perfectly. How do you get the shaders into your OpenGL program?

Assuming you're using Xcode, add the shaders to your project. You don't have to add the OpenGL Shader Builder project file. Just add the shaders. Xcode can't compile shaders. You want Xcode to copy the shaders to the application bundle when it builds your project, just like graphics and audio files. Make sure the shaders are in the target's Copy Bundle Resources build phase. Chapter 6, "Building Projects", has detailed information on build phases, specifically the "Target Build Phases" section.

After adding the shaders to your project, you have two coding tasks in your OpenGL program. The first task is to create the shader. The second task is to create a program object and attach the shader to it. I'm using the OpenGL 2.0 syntax. If you're using ARB extensions to support older graphics cards, the syntax may differ slightly. OpenGL 3 and OpenGL ES 2.0 use slightly different syntax, which mostly involves removing the `gl` prefix from each function name. OpenGL ES also removes the `GL` prefix from constants and data types.

Creating a Shader

Creating a shader requires the following coding tasks:

1. Create a shader object by calling `glCreateShader()`.
2. Load the shader from disk.
3. Read the shader source code by calling `glShaderSource()`.
4. Compile the shader by calling `glCompileShader()`.

Creating a Shader Object

Call the function `glCreateShader()` to create a shader object. The function `glCreateShader()` returns an integer value. Supply one of the following values:

- `GL_VERTEX_SHADER` for a vertex shader.
- `GL_FRAGMENT_SHADER` for a fragment shader.
- `GL_GEOMETRY_SHADER` for a geometry shader.

The following code creates a vertex shader:

```
GLuint shader;  
shader = glCreateShader(GL_VERTEX_SHADER);
```

Loading a Shader

How you load the shader from disk depends on the application framework you're using. If you're using Cocoa, you would use Cocoa's `NSBundle` class to load the file from the application bundle. The following code demonstrates how to load a shader file from the application bundle:

```
NSString* filename;
NSString* extension;

NSBundle* appBundle = [NSBundle mainBundle];
NSString* shaderFile = [appBundle pathForResource:filename
                                     ofType:extension];

// Convert the NSString to a C String for OpenGL to use
const GLchar* shaderFileGL;
shaderFileGL = (const GLchar*)[shaderFile UTF8String];
```

Reading Shader Source

Call the function `glShaderSource()` to read the shader source. `glShaderSource()` takes four arguments.

- The shader index you created by calling `glCreateShader()`.
- The number of strings, which would be 1 to load a single shader.
- The string, which is the shader file you loaded from disk.
- The string length, which you can set to `NULL` if you don't care about the length.

The following code loads a single shader that was loaded from disk:

```
GLuint shader;
const GLchar* shaderFileGL;

glShaderSource(shader, 1, &shaderFileGL, NULL);
```

Compiling the Shader

To compile the shader, call the function `glCompileShader()`. Calling `glCompileShader()` is relatively easy. Supply the shader index you created by calling `glCreateShader()`.

```
glCompileShader(shader);
```

Creating a Program Object

Creating a program object takes four steps.

1. Create the program object by calling `glCreateProgram()`.
2. Attach the shader to the program object by calling `glAttachShader()`.
3. Link the program by calling `glLinkProgram()`.
4. Use the program object in your application by calling `glUseProgram()`.

The following code demonstrates the creation of a program object:

```
GLuint program;
GLuint shader;

program = glCreateProgram();
glAttachShader(program, shader);
glLinkProgram(program);
glUseProgram(program);
```

Cleanup

When you're done with a shader, there are three tasks to perform. The first task is to detach the shader from the program object by calling `glDetachShader()`. This function takes two arguments: the program object and the shader.

```
glDetachShader(program, shader);
```

The second task is to delete the shader by calling `glDeleteShader()`. Supply the shader to delete.

```
glDeleteShader(shader);
```

The final task is to delete the program object by calling `glDeleteProgram()`. Supply the program object to delete.

```
glDeleteProgram(program);
```

OpenGL ES Performance Detective

OpenGL ES Performance Detective investigates your OpenGL ES application to see if its frame rate is limited by the graphics pipeline. If the frame rate is limited by the graphics pipeline, OpenGL ES Performance Detective lists the most likely causes of the graphics pipeline performance issues.

Apple integrated OpenGL ES Performance Detective into Xcode's debugger in Xcode 4.5. There is no separate OpenGL ES Performance Detective application in Xcode 4.5 and later.

Before You Run Performance Detective

There are two things you must do before you run an application with OpenGL ES Performance Detective. First, you must place the application on the device. Second, you must connect the device to your Mac and tell Xcode to use it for development. Open the Organizer, click the Devices button at the top of the window, and select the device on the left side of the Organizer. Click the Use for Development button.

Running Performance Detective

When you launch OpenGL ES Performance Detective, a welcome screen opens. Click the Open New Case button. Use the pop-up menus to choose a device and an application to trace. If you need to supply command-line arguments, click the disclosure triangle next to Arguments. Click the Open Case button.

When you click the Open Case button your application launches on the device. OpenGL ES Performance Detective displays the frame rate for your application. Click the Collect Evidence button when you are finished profiling.

You can also launch OpenGL ES Performance Detective from Xcode by using the scheme editor. Select the Profile action in the scheme editor. Choose OpenGL ES Performance Detective from the Instrument pop-up menu. Choose Product > Profile. OpenGL ES Performance Detective launches and starts running your application.

Viewing the Results

Clicking the Collect Evidence button opens the trace window. Every trace contains a summary. The summary tells you whether or not your application's frame rate is limited by the graphics pipeline. If the frame rate is limited by the graphics pipeline, the trace window displays the top suspects. For each suspect OpenGL ES Performance Detective provides a description of the problem as well as possible fixes.