Fritz Anderson

Xcode 3

UNLEASHED

Xcode 3 Unleashed

Copyright © 2009 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-321-55263-1

ISBN-10: 0-321-55263-6

Library of Congress Cataloging-in-Publication Data:

Anderson, Fritz.
Xcode 3 unleashed / Fritz Anderson. — 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-55263-1 (pbk. : alk. paper)
ISBN-10: 0-321-55263-6 (pbk. : alk. paper) 1. Operating systems (Computers)
2. Macintosh (Computer) I. Title.
QA76.76.063A53155 2009
005.4'32—dc22

2008017851

Printed in the United States on America

First Printing: August 2008

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Que Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Apple, the Apple logo, Cocoa, Finder, Macintosh, Mac OS, Mac OS X, Objective-C, and Xcode are registered trademarks of Apple, Inc.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to http://www.quepublishing.com/safarienabled.
- Complete the brief registration form.
- Enter the coupon code ARHL-WXMH-HFKV-DHHN-287M.

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please email customer-service@safaribooksonline.com.

Editor-in-Chief
Karen Gettman

Senior Acquisitions Editor
Chuck Toporek

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Keith Cline

Indexer
Cheryl Lenser

Proofreader
Leslie Joseph

Publishing Coordinator
Romny French

Multimedia Developer
Dan Scherf

Cover Designer
Gary Adair

Compositor
Nonie Ratcliff

Preface

Xcode is the central tool for developing software for Mac OS X. It was my privilege to help explain that tool in *Step into Xcode: Mac OS X Development*. Since then, Apple has released a new operating system, Leopard, and a new Xcode. Xcode 3 is the official development tool for Apple's iPhone. *Xcode 3 Unleashed* is a new edition for a new world.

I wrote *Xcode 3 Unleashed* for people who are new to Mac programming and to Xcode, but I've included plenty of material that will be new even to experienced developers. My approach is to lead you through a simple application project to give you a vocabulary for the workflow of Mac development, and how Xcode and the tools that accompany it fit in. After you have a solid grounding, we can move on to Part II, where the details and more advanced techniques can come out.

Part I is a practical introduction, showing how to use Xcode at every step, from building a command-line tool, to debugging, to building a human interface, to Core Data design and language localization. Companion tools such as Interface Builder and Instruments are essential to developing for the Mac, and I cover them.

Version control has become indispensable even to small, single-programmer projects. *Xcode 3 Unleashed* introduces you to source-code management early, and returns to it frequently.

Part II covers how to use Xcode to manage and navigate your code base, even if it comes from a large, open source UNIX project. It shows how Xcode's build system—the mechanism that decides how and when to turn your code into an application—works. I return to Instruments, the astonishing tool for timelining your programs' execution and use of resources, and introduce Apple's performance tools, led by the deep and powerful Shark statistical profiler.

Version Covered

I started writing *Xcode 3 Unleashed* when Xcode 3.0 was in development. 3.0 was the version in general release when we went to press, although Apple had started a beta program for version 3.1, under nondisclosure. There are many improvements in 3.1, but none that significantly change this book's lessons.

Where I found bugs or feature gaps in Xcode 3.0, I noted them. If you're using a later version, you might find those bugs have been cleared. Apple's Xcode team continues to work hard on the developer tools.

Typographic Conventions

Xcode 3 Unleashed uses a few conventions to make the material easier to read and understand.

NOTE

Notes are short comments on subjects that relate to the text, but aren't directly in the flow.

WARNING

Warnings raise points that might trip you up, commit you to a dead end, or even make you lose your work.

Sidebars

Sidebars are for extended discussions that supplement the main text.

- `Monospaced` type is used for programming constructs, filenames, and command-line output.
- Text that you type is shown in **`monospace bold`**.
- Human interface elements, such as menus and button labels, are shown **like this**.
- When new terms are introduced, they are set off *in italics*.
- And program listings are shown in the colors you would see in Xcode's editors.

The Mac keyboard provides four modifier keys, and Xcode uses them all liberally as shortcuts for menu commands. This book denotes them by their symbols as used in the menus themselves:

Command	⌘
Shift	⇧
Control (Ctrl)	ˆ
Option (Alt)	⌥

Introduction

IN THIS INTRODUCTION

- What Xcode Is
- What's New in Xcode 3
- Obtaining Xcode
- Installing Xcode

From the moment it first published Mac OS X, Apple, Inc., has made a complete suite of application development tools available to every user of the Macintosh. Since Mac OS X version 10.3, those tools have been led by Xcode, the integrated development environment Apple's own engineers use to develop system software and applications such as Safari, iTunes, Mail, and iChat. If you own a Mac, these same tools are in your hands today.

What's New in Xcode 3

In October 2007, with the introduction of Mac OS X 10.5 (Leopard), Apple introduced version 3 of the Xcode developer tools suite. Among the changes were

- Extensive improvements to the Xcode integrated development environment (IDE), including
    - Support for Objective-C 2.0, the first major revision to the language, with commands for converting existing code to the new language.
    - Improved syntax coloring, now including distinctive colors for symbols like instance variables and method names.
    - Code Focus, a ribbon beside the editor text that lets you see how blocks of code are organized, and allows you to fold long blocks down to the height of a single line.
    - The projectwide **Find** command now works through the Spotlight text-searching engine, yielding better results faster.

    - A debugger bar, offering simple debugging controls in any editor window.
    - Datatips, allowing you to inspect the values of program variables during debugging, just by hovering the cursor over them in the code.
    - A mini-debugger, injected into the programs you run, permitting debugging during mouse-down events and other "volatile" situations.
    - Automatic access to the debugger whenever a program you run from Xcode crashes.
    - Improved compile-time error reporting, interleaving compiler messages with the code they relate to.
    - Automated refactoring, helping you rename classes, methods, and functions, shift methods from class to class, and even create new super classes, in an Objective-C project.
    - Much improved support for source code management tools such as Subversion, CVS, and Perforce.
    - Much improved support for using UNIX scripting languages to create and edit text.
    - The Organizer, a window to hold references to frequently used files and projects.
    - Among the command-line tools, the new `xed` tool enables you to open text files in Xcode, when a shell script or tool demands an interactive editor.
    - A major upgrade to the documentation system, using RSS feeds for live updates, and permitting developers to add their own documentation to the system.
    - A Research Assistant window that documents API symbols and build variables in real time, as they are selected.
- A completely revamped Interface Builder, with better tools for crafting nonvisual parts of the human interface, such as controller objects. Integration between IB and Xcode is even tighter than before.
- A new tool, Instruments, for profiling the resource usage (memory, I/O, graphics, threading) of a program, in real time, on a timeline so that you can see how each element of the performance picture relates to all the others.

Xcode 3 is a ground-up rebuild of the Mac OS X developer tools, and it has been well worth the wait.

Obtaining Xcode

If you have an installation DVD for Mac OS X 10.5 or a new Mac on which Leopard has come installed, you already have Xcode. On the DVD, an installation package can be found in the `Xcode Tools` folder inside the `Optional Installs` folder. On new Macs, you'll find a disk image file for Xcode Tools in the `Additional Installations` folder at the root of your hard drive; double-click the disk image to mount it, and you'll find the installation packages inside.

However, Apple does not always coordinate the latest version of its developer tools with its Mac OS X distributions. Even if you have an installation package on your Mac, or on your distribution disk, it pays to check for a newer version at the Apple Developer Connection (ADC).

Downloading Xcode

You must join ADC to download Xcode. Point your web browser to http://developer.apple.com/, and click the link that offers a membership (at the time of this writing, it was the **Sign Up** link at the top of the page). You will be offered a handful of options, some expensive. All you need is an Online membership—it's free. Fill out the forms offered to you; they will take contact information and ask you to consent to terms and conditions. There may be marketing questions and offers of mailings.

When you have completed the signup process, go to http://connect.apple.com. Fill in the username and password you chose. You will then be presented with a few options, among these being **Downloads**. This is what you want; click it.

Depending on your membership level, and how active Apple has been lately in releasing new software, you might not be able to find Xcode on this page. If you don't see it, click **Developer Tools** in the Downloads column at the right of the page. Scroll down to the first Xcode 3.x download you find (earlier releases may appear lower in the list, and versions of Xcode 2.5 may appear higher). It will be a disk image a bit over 1GB in size. This will comprise the full set of Xcode tools; there is no updater you can apply to a copy you may already have. Click to download.

Installing Xcode

Now that you have the latest Xcode package, it's time to install it. Installation packages can be run straight from a DVD, a mounted disk image file, or your hard disk. There's no difference.

In the `Xcode Tools` folder, you will find three installation packages:

- `XcodeTools.mpkg`, which is the installation package for Xcode and the other tools needed for Mac OS X development.
- `Dashcode.mpkg` provides the Dashcode IDE for producing Dashboard widgets. Dashcode is also included in the standard install from `XcodeTools.mpkg`; this package is for those who are interested only in developing widgets.

- `WebObjects.mpkg` installs Apple's excellent WebObjects frameworks and tools, for developing sophisticated database-centered websites in Java. WebObjects is also available as an optional install from within the Xcode Tools Installer.

You will also find a folder named `Packages`, containing installation packages for components of the Xcode tools, like the CHUD performance-measuring suite, software development kits (SDKs) for X Window and earlier versions of Mac OS X, and version 3.3 of the `gcc` compiler suite (for PowerPC Macs only). All these are available as options (or within options) in the Xcode Tools Installer, but are here in case you omit them from the original installation and want to add them later.

If you've ever done an installation under Mac OS X, the Xcode tools install is familiar (see Figure I.1). Start by double-clicking the `XcodeTools.mpkg` installation package. A Welcome screen appears, at which you will press **Continue**. Next, the installer displays the license for Xcode and its related software; click **Continue**, and if you accede to the license, click **Agree** in the ensuing sheet.

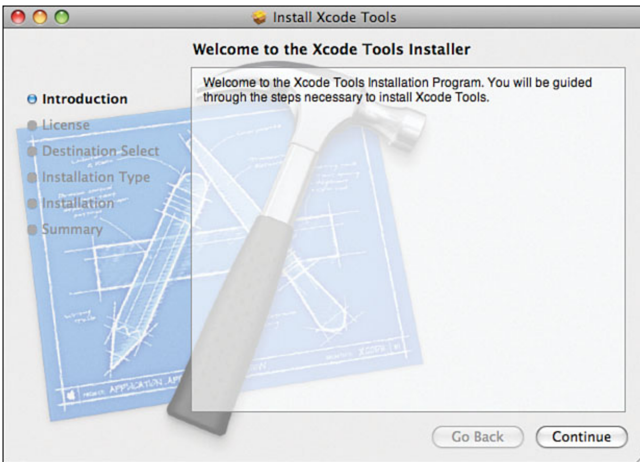

FIGURE I.1 The Welcome panel for the Xcode Tools Installation package should be familiar to any experienced Mac user.

You are now at the Standard Install panel, but we will vary from the standard line. Click the **Customize** button to reveal the Custom Install panel. This panel (see Figure I.2) contains a table listing the components of the Xcode Tools installation. The single mandatory component is checked and grayed out; the optional components are active, and you can check or uncheck them to include or exclude them from the installation:

- **Developer Tools Essentials**. This is Xcode itself, and the graphical and command-line programs that complement it, plus SDKs for developing Mac OS X software for versions 10.4 and later. This is a mandatory component; it doesn't make sense to install the developer tools without installing Xcode and the tools needed for it to run.

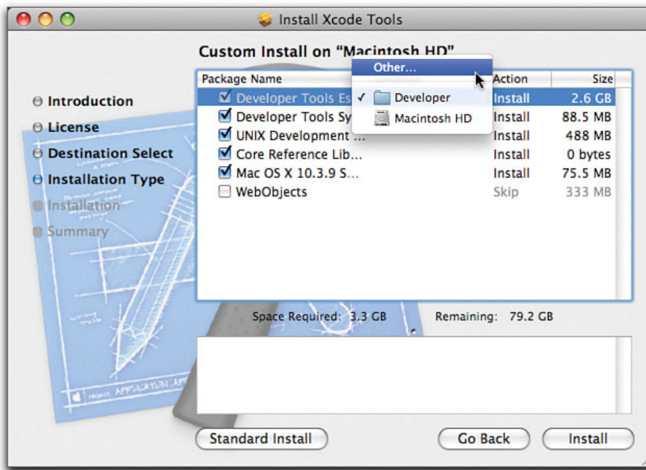

FIGURE I.2 The Custom Install panel for the Xcode Tools Installer. The top entry in the package list is for the core Xcode tools, and is not optional. In the `Location` column is a pop-up menu from which you can select where the developer tools are to go; the default is the `Developer` folder of your boot disk.

- **Developer Tools System Components**. These are the CHUD tools for investigating application performance, plus facilities for distributing application builds over more than one computer. You should install this package.

- **UNIX Development Support**. The "essentials" installation of Xcode installs components such as compilers and their support files in a `usr` subdirectory of the installation directory. If you will be doing command-line development—for instance, for building open source projects—you will want a set of development tools installed in the root `/usr` directory tree. This package installs copies of the command-line tools into `/usr`. You should install this package; examples in this book depend on it (see Chapter 24, "A Legacy Project").

- **Core Reference Library**. This package installs the panoply of introductions, references, technical notes, and sample code that document development on Mac OS X and the APIs you need to do it. Install this package.

- **Mac OS X 10.3.9 Support**. Installs the SDK and tools needed to produce software that targets Panther (Mac OS X 10.3). This includes version 3.3 of the `gcc`

NOTE

CHUD, `gcc` 3.3, and WebObjects are not flexible about where they are installed. They will be installed into `/Developer` no matter what location you choose for the Xcode tools.

NOTE

The `gcc` compiler suites installed with the Xcode tools are Apple-modified builds that take account of such Mac OS X features as frameworks and support for Objective-C 2.0. They are not the same as the `gcc`s available under the same version numbers from the Free Software Foundation.

compiler suite, for PPC Macs only. This package is left out of the standard install, and whether you need it depends on whether you intend to build applications for 10.3 (see Chapter 17, "Cross-Development").

- **WebObjects**. This package installs the applications and files needed to develop web applications with Apple's WebObjects framework. You need not install this package.

Unlike earlier versions, Xcode 3 and 2.5 are flexible about where you install them. This is where you would make that choice. See the section "Another Install Location" for details.

NOTE

Earlier versions of Xcode offered to install reference material for the current Java development kits. These are still available through the Downloads section of ADC.

NOTE

With Xcode 3, Apple has dropped support for developing software aimed at Jaguar (Mac OS X 10.2). If you need to target 10.2, you must install Xcode 2.5 and use its 10.2.8 SDK. You cannot use Xcode 2.5 to build software using 10.5 technologies.

Now click the **Install** button. The standard authentication sheet will appear, into which you enter the name and password of an administrative user of your Mac.

The next panel contains a progress bar and a narrative of what is being installed. This process takes a number of minutes, at the end of which you are rewarded by a big green check mark. Close the installer; you are now ready to use Xcode.

Another Install Location

Earlier versions of Xcode (earlier than 2.5) installed themselves only in the `/Developer` directory of the startup file system. Having one possible path to all the developer tools greatly simplified the task of locating them: If you needed the `packagemaker` tool, it was at `/Developer/Tools/packagemaker`, and that was that.

Things have changed since then. First, the Xcode package has grown larger and larger. The download package alone is 1.1GB in size, which expands to 3.3GB installed. It is reasonable to want to put Xcode onto another disk or partition. Second, it is now possible to install Xcode 2.5 (see the section "Xcode 2.5" that follows) in parallel with Xcode 3, and the two tool sets necessarily need two homes.

That is why, during your installation of Xcode, `/Developer` is still the default location, but you can choose another.

If you want another location, ignore the **Change Install Location** button on the Standard Install panel. The Installer application offers this button as a standard part of its workflow; if you press it, you will find that the boot volume is your only choice. If you find yourself at the Install-Location panel, click the **Go Back** button to back out.

The real choice comes in the Custom Install panel. In the Standard Install panel, click the **Customize** button to get to a list of components to install. The top line, **Developer Tools**

Essentials, has a pop-up menu for setting the location for installing the Xcode tools (see Figure I.2). The default location, `/Developer`, is shown initially.

Change the location by selecting **Other** from the pop-up. A standard open-file sheet will appear. Find the directory that you want to contain your Xcode directory. Use the **New Folder** button to create the Xcode directory there. Make sure that directory is selected, and then click **Choose**. The selected directory will contain the `Applications`, `Documentation`, and other directories that make up the Xcode tool set.

NOTE

Yes, you can put Xcode wherever you want, but accounting for that possibility in every reference to a component of the developer tools would make this book more tedious than it has to be. I'll just refer to the Xcode tools directory as `/Developer`, and trust you to make the transposition yourself.

You can continue the installation from there, as before.

Uninstalling Xcode

All things come to an end, and there is no exception for Xcode. There are two reasons you might choose to remove Xcode from your hard drive. The first is that you just do not want it; you want the files gone, and the space reclaimed.

The second is that you want to install a later version of Xcode. When Apple comes out with new versions of Xcode, it does not distribute updaters. Only the full Xcode tools package will be available for download. Past experience has shown that a full upgrader is a bigger and more accident-prone undertaking than the Xcode team can sustain, especially when the alternative is to have Xcode users simply remove the earlier version and install the new version afresh.

The developer tools come in two parts. The most prominent is the `/Developer` directory itself, which contains all the graphical applications, documentation, and SDKs that make up the public face of Xcode. The other part is the tools embedded throughout the UNIX file system that make development possible. For instance, two versions of the `gcc` compiler are installed at `/usr/bin`; all the headers needed for development on the current system are in the huge `/usr/include` hierarchy. To properly uninstall the developer tools, these, too, have to be picked through and removed.

The first part of the uninstallation is easy: Find your Xcode tools directory, and drag it into the trash. That's 100,000-plus files gone.

Next, execute the tool `/Library/Developer/Shared/uninstall-devtools` from the command line. `uninstall-devtools` is a Perl script that walks through the saved installation receipts looking for every developer tools package going back to 2001. It deletes the files of every package it finds. Running `uninstall-devtools` will take a few minutes. At the end, you have a system fit for a fresh install.

This procedure is good enough if you mean to reinstall the developer tools. If you mean to go further, you also want to delete the directories `/Library/Developer` and

`~/Library/Developer`, and the preference files for the individual developer applications. The usual procedure spares these, because they contain customization files you may have created, which you would want to carry over to a new installation.

Xcode 2.5

Many people have commitments to Xcode 2 that they can't get out of, even if they are running Leopard. Managers of a project nearing completion, with many developers, may be reluctant to revalidate their build processes for a new tool chain.

They might have NIBs that rely on palettes for Interface Builder 2, which are not usable in IB 3. Further, although Xcode and Interface Builder do provide "compatibility" modes, it is easy to produce files that earlier versions cannot open. Holding off on Xcode 3, at least for some projects, can be prudent.

That is why Apple released, in parallel with Xcode 3, Xcode 2.5. The Xcode 2.5 tools are strictly file compatible with those of the preceding version, Xcode 2.4. Unlike version 2.4, 2.5 can run on either Tiger (Mac OS X 10.4) or Leopard (10.5).

Like Xcode 3, Xcode 2.5 permits you to choose where to install its developer tools. As with the Xcode 3 installation, you are offered a **Customize** button for editing the components to be installed. The top component, representing the core developer tools, will have a pop-up enabling you to chose where to install Xcode 2.5. The default location is `/Xcode2.5`.

If you intend to develop specifically for Mac OS X 10.5, Xcode 2.5 is not for you; it does not support the Leopard SDK. For Leopard development, you have to use Xcode 3.

If you have Project Builder (`.pbxproj`) projects around, now is the time to convert them to Xcode projects, and 2.5 is the tool to do it. Xcode 3 has dropped the capability to import Project Builder projects.

Having two Xcodes on your system gives you two versions of Xcode-related command-line tools such as `xcodebuild`. If you opt (as I strongly recommend) to install tools in `/usr/bin`, it is a nice question which version of a tool is run when you execute it from the command line or a build script. The solution is this: The `/usr/bin` versions of these tools are in fact scripts that refer to the binary versions in the Xcode 3 or 2.5 install tree. You determine which version is used by running the `xcode-select` tool; `man xcode-select` for details.

CHAPTER 26
Instruments

IN THIS CHAPTER

- Instruments: Performance and Resources in Time
- Using the Instruments Window
- Configuring Instruments
- Apple's Templates and Instruments
- Custom Templates and Instruments

Instruments is a framework for software-monitoring tools called… instruments. (Capital *I* Instruments is the application, small *i* instruments are components of the Instruments application.) The analogy (borrowed from Apple's Garage Band audio editor) is to a multitrack tape deck. Instruments records activity on one or more tracks (one per instrument), building the data on a timeline like audio on a tape.

We've seen Instruments before, in Chapter 19, "Finishing Touches," where it helped us track down a memory leak in Linear. It deserves a chapter all its own.

What Instruments Is

The focus on a timeline makes Instruments unique. We saw how MallocDebug collects allocation and deallocation events, and gathers them into statistical measures, organizing all the stack traces it found at those events into an aggregate call tree, from which you can learn how memory is used. It presents data as an end-of-run accumulation.

Shark, too, works by statistical aggregates. You run your application, Shark samples it, and in the end it presents you with profiling information that is a summary (although very detailed) of all the samples of the whole run. You can filter the samples and manipulate the call trees Shark reports, but the product is still a compilation over a period of time. There *is* a chart view, but it is still an aggregate, showing the shape of the call stack over time. You can examine stack traces to see what the processor was doing at the time (it can be tricky to select exactly the right one), but there is no way to relate the traces to what the *application* was doing.

Further, tools such as MallocDebug and Shark do one thing at a time. MallocDebug does heap memory. Shark does profiling (or `malloc` tracing, or processor events). If you want a different measure, run the application again under the supervision of a different tool. They allow no way to see what one measure means in relation to another.

Instruments is different. It is comprehensive. There are instruments for most ways you'd want to analyze your code, and Instruments runs them *all at the same time*. The results are laid out by time, in parallel. Did clicking the **Compute** button result in Core Data fetches? Or had the fetches already been done earlier? Did other disk activity eat up bandwidth? In the application? Elsewhere in the system? Is the application consuming too many file descriptors, and if so, when, and in response to what? You're handing data off to another process (think `Linrg`, from the first iteration of `Linear`); how does the tool's memory usage change in response to the handoff, and how does it relate to the use of file descriptors in both the tool and the master application?

Instruments can answer these questions. You can relate file descriptors to disk activity, and disk activity to Core Data events, with stack traces for every single one of these, because Instruments captures the data on a timeline, all in parallel, event by event. And, you can target different instruments on different applications (or even the system as a whole) at the same time.

NOTE

Most of the power of Instruments lies in the analysis tools it provides after a recording is made, but don't ignore the advantage it provides in showing program state dynamically: If you can't see when memory consumption or file I/O begins and settles down (for instance), you won't know when to stop the recording for analysis in the first place.

Running Instruments

In Chapter 19, we started Instruments from Xcode by selecting **Run** > **Start with Performance Tool** and selecting an **Instruments** template. At least as often, you'll just launch the Instruments application from the `/Developer/Applications` directory.

When you start it, Instruments automatically opens a document (called a *trace document*) and displays a sheet offering you a choice of templates populated with instruments for common tasks (see Figure 26.1). You can find a complete list of the templates Apple provides in the section "The Templates" later in this chapter.

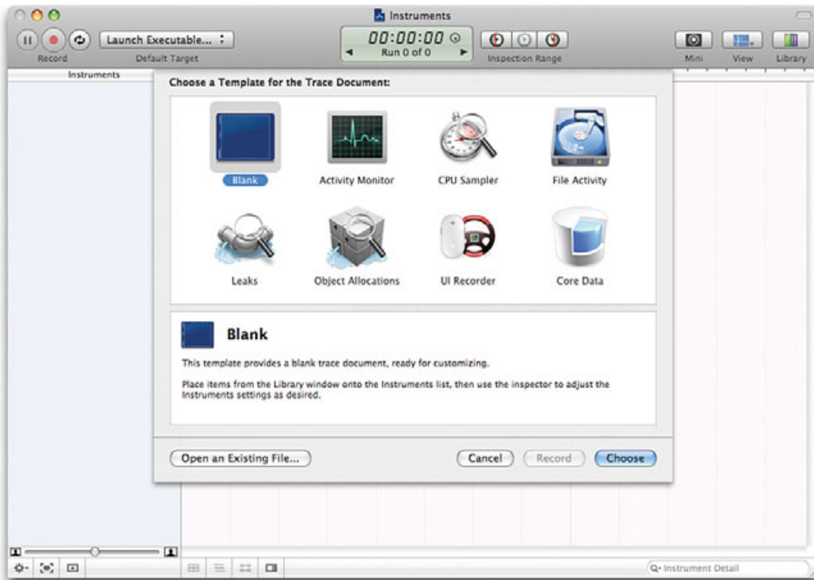

FIGURE 26.1 When you create a new trace document in Instruments, it shows you an empty document and a sheet for choosing among templates prepopulated with instruments for common tasks.

The Trace Document Window

The initial form of a trace document window is simple: a toolbar at the top, and a stack of instruments in the view that dominates the window. After you've recorded data into the document, the window becomes much richer. Let's go through Figure 26.2 and identify the components.

The Toolbar

The toolbar comes in three sections. The controls at left (1) control recording and the execution of the target applications. There is a pause button for suspending and resuming data collection, a **Record / Drive & Record / Stop** button to start and stop data collection, and a loop button for running a recorded human-interface script repeatedly.

> **NOTE**
>
> When you start recording, you will often be asked for an administrator's password. The kind of deep monitoring many instruments do is, strictly speaking, a security breach, and the system makes you show you are authorized to do it.

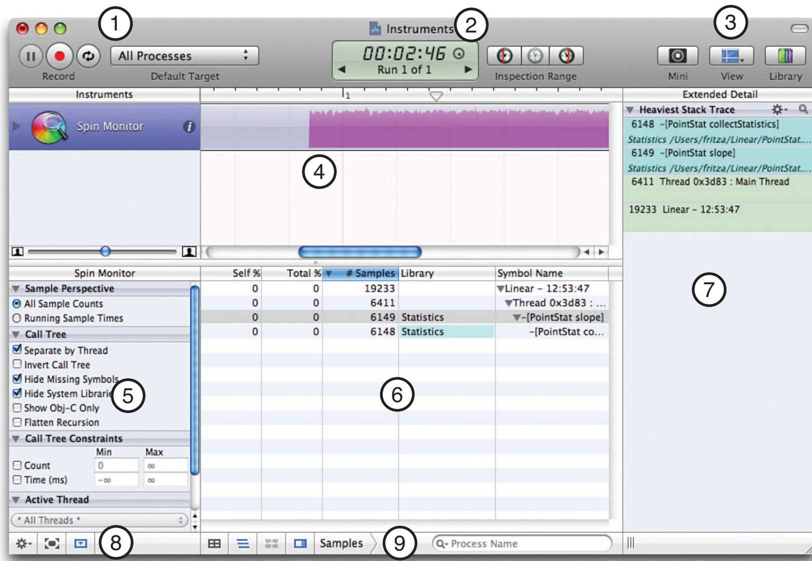

FIGURE 26.2 A typical Instruments window, after data has been recorded. The Extended Detail pane (at right) has also been exposed. I discuss the numbered parts in the text.

The **Default Target** pop-up designates the process or executable that all instruments in the document will target, unless you specify different targets for individual instruments. The choices are as follows:

- **All Processes**. Data will be collected from all the processes, user and system, on the machine. For instance, the Core Data instruments can measure the Core Data activity of all processes. Not every instrument can span processes; if your document contains no instruments that can sample systemwide, this option is disabled.

- **Attach to Process**. Data will be collected from a process that is already running; select it from the submenu. Some instruments require that their targets be launched from Instruments, and cannot attach to running processes. If you use only nonattaching instruments, this option is disabled.

- **Launch Executable**. When you start recording, Instruments will launch the selected application or tool, and collect data from it. The submenu contains items for applications you've recorded previously, and has a **Choose Executable** item to select a fresh application.

- **Instrument Specific**. Each instrument will collect data from the target specified in the **Target** pop-up of its configuration inspector. The instruments in a trace document do not all have to collect data from the same target.

The center section (2) relates to time (see Figure 26.3). The clock view in the center of the toolbar displays the total time period recorded in the document. If you click the

clock-face icon to the right of the time display, the clock shows the position of the "playback head" in the time scale at the top of the Track pane.

The clock view also controls which run of the document is being displayed. Each time you click **Record**, a new recording, with a timeline of its own, is added to the document. The run now being displayed is shown like "Run 1 of 2," and you can switch among them by pressing the arrowhead buttons to either side.

Most instruments will display subsets of the data they collect if you select a time span within the recording. To do so, move the playback head to the beginning of the span, and click the button on the left of the **Inspection Range** control; then move the head to the end of the span and click the button on the right. The selected span will be highlighted, and the Detail pane will be restricted to data collected in the span. To clear the selection, click the button in the middle.

FIGURE 26.3 The center section of a trace document's toolbar displays a clock, and controls for selecting a span of time within a recording. The clock displays the total time in the document (or, if you click the icon at the right of the clock, the position of the "playback head") and the run being displayed if there is more than one.

NOTE

You can also browse among runs by selecting **View > Run Browser** (**^Tab**). The contents of the window will be replaced by a "Cover Flow" partial view of the traces in each run, along with particulars of when it was run, on what machine, and so on.

NOTE

Option-dragging across an interval in one of the traces will also set an inspection range.

The right section (3) provides convenient controls for display. **Mini** hides Instruments and displays a heads-up window for controlling recording from other applications. **View** pops up a menu that shows and hides the Detail and Extended Detail panes. **Library** shows and hides the Library window.

The Track Pane

The Track pane (4) is the focus of the document window, and the only component you see when a document is first opened. This is the pane you drag new instruments into. Each instrument occupies its own row, with a configuration block on the left, and the instrument's track on the right.

The configuration block (see Figure 26.4) shows the instrument's name and icon. To the left is a disclosure triangle so you can see the instrument's track for each run in the document. To the right is an inspector button (**i**) that reveals a configuration inspector for the instrument.

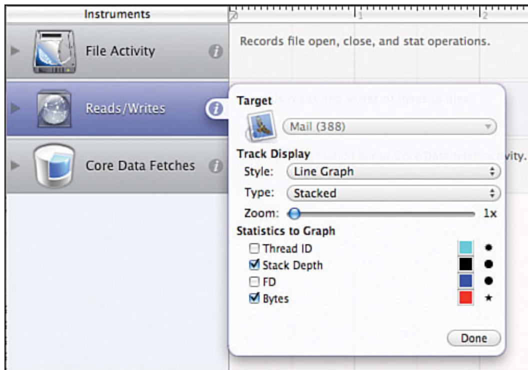

FIGURE 26.4 A stack of instrument tracks in a trace document. Each instrument has its own row, with a timeline extending to the right, calibrated in seconds. Clicking an instrument's configuration button opens an inspector containing settings for the instrument. Some of these control the style of the graph and which of its data an instrument displays, and can be changed at any time. The pop-up at the top of the inspector selects what process the instrument is to collect data from, and must be set (or left to the default, if the instrument can accept it) before recording begins.

The tracks to the right of the configuration blocks display the data collected by the instruments, on a timeline. The configuration inspector controls what data is plotted, and how it is displayed.

At the top of the timeline is a ruler matching the data to the time at which is was collected. The scale of the track can be controlled by the slider below the configuration blocks. In the ruler, you will see a white triangle, the *playback head*. Drag the playback head and use the **Inspection Range** control to select intervals within the recording. As you drag the head across the track, many instruments will label their tracks with the value of their data at that time.

The Detail Pane

The Detail pane (5 and 6) appears when you've made a recording, or when you use a control or menu item to display it. **View** > **Detail** (⌘D), **View** (toolbar button) > **Detail**, the **Detail** item in the **Action** (⚙) menu at the lower left of the window, and the detail button (rectangle with arrowhead) next to it, will all toggle the Detail pane. You do not lack for options.

When you select an instrument in the Track pane, the data from the instrument collection is shown in tabular form in the Detail pane. What's in the table varies among instruments. And, the Detail pane controls (9, and Figure 26.5) provide for up to three table formats.

Like the table itself, the alternative views vary depending on the instrument. The general pattern seems to be

FIGURE 26.5 The Detail View buttons, which appear below the table portion of the Detail View. The first three buttons select different table displays, or "modes." The modes are Table, Outline, and Diagram. The fourth button, showing a window with a portion on the right highlighted, opens or closes the Extended Detail pane. The Navigation Path breadcrumb control enables you to back a display off after you've drilled down into a detail.

- Table mode is the principal display the instrument's author has chosen for its data. For most instruments, this is the raw data they collected, such as the details of individual calls in the Reads/Writes instrument. In Sampler, the table contains a stack trace for each sample; in ObjectAlloc, the items are classes/categories of allocated blocks.
- Outline mode, in the case of instruments that collect stack traces, aggregates the traces into call trees (like the Tree and Heavy displays in Shark). When this is the case, the **Call Tree** controls in the Detail controls view (5) become active.
- Diagram mode is not often used. In Chapter 19, we saw that ObjectAlloc used this for a table of the individual data it collects.

The next button over, with an icon that suggests a window with a portion highlighted, displays the Extended Detail pane (7), which is covered in the next section.

When you "drill into" data in a Detail pane, such as when you obtain the history of an allocated block in ObjectAlloc, the "breadcrumb" control at the right end of the Detail controls enables you to back out to the superior view.

The left portion of the Detail pane (5) contains controls to adjust or analyze the contents of a Detail table, and in some cases to configure an instrument before it is run. The repertoire of controls varies by instrument and view, but the most commonly used controls are in the group labeled **Call Tree**, which is active whenever a tree of call stacks is displayed in the table.

These commands are similar to the stack data-mining options available in Shark:

- **Separate by Thread**. Call trees are normally merged with no regard for which thread the calls occurred in. Separating the trees by thread will help you weed out calls in threads you aren't interested in.
- **Invert Call Tree**. The default (top-down) presentation of call trees starts at the runtime `start` function, branching out through the successive calls down to the leaf functions that are the events the instrument records. Checking this box inverts the trees so that they are bottom up. The displayed tree begins at the "event" function, and branches out among its callers, thus aggregating call paths to bottleneck functions.
- **Hide Missing Symbols**. Checking this box hides functions that don't have symbols associated with them. If you can't determine what they are, they probably aren't part of your code. (If they are part of your code, turn off symbol stripping in your build.)

- **Hide System Libraries**. This skips over functions in system libraries. Reading the names of the library calls may help you get an idea of what is going on; if you are looking for code you can do something about, however, you don't want to see them.
- **Show Obj-C Only**. Checking this narrows the list down to calls made from Objective-C methods, whether in system libraries or not (another way to cut out the possible distraction of calls you don't care to see).
- **Flatten Recursion**. This lumps every call a function makes to itself into a single item. Recursive calls can run up the length of a call stack without being very informative.

You can also add call-tree constraints, such as minimum and maximum call counts. The idea is to prune (or focus on) calls that are not frequently made. Another constraint that may be available (for instance in the Sampler instrument) can filter call trees by the amount of time (minimum, maximum, or both) they took up in the course of the run.

> **NOTE**
>
> Stack traces in the Extended Detail pane reflect your settings of these filters.

Of course, another way to filter call trees is to restrict your attention to a particular time span, such as between the `open` and `close` calls on a particular file (which the File Activity instrument would landmark for you). Use the playback head and the **Inspection Range** control to select the beginning and end of the period of interest, and the call tree will reflect only the calls made between them.

The Extended Detail Pane

The Extended Detail pane (7) typically includes a stack trace when you select an item in the Detail pane that carries stack information. When the selected item is part of a call tree, the Extended Detail pane shows the "heaviest" stack, the one that accounts for the most of whatever the instrument keeps track of. Selecting a frame in the call stack highlights the corresponding call in the call-tree outline. Double-clicking a frame opens the corresponding source code in Xcode, if it can be found.

A stack trace in the Extended Detail pane has an **Action** (⚙) menu at the top. Most commands in this menu have to do with how the calls in the trace are displayed. An example is **Color by Library**, which tints each call frame by the library file (including an application's main executable) that the call came from.

> **NOTE**
>
> The Extended Detail pane can include other information. There may be a **General** item, summarizing the information in the item selected in the Detail table, or a **Time** item showing how far into the recording the selected event occurred. Because the stack trace is only one item, its **Action** menu is attached to the divider bar that marks it. Scrolling down the stack trace may scroll the **Action** menu out of sight.

A couple of items in the **Action** menu are of particular interest. **Look Up API**

Documentation acts like Option-double-clicking a symbol in Xcode: Select the frame, select the command, and be directed to its documentation in Xcode's Documentation window. **Trace Call Duration** creates a new instrument in the current document to record the stack trace when the function was called, and how long it took to execute.

Controls

Three additional controls are to be found at the bottom-left corner of the document window (8) (see Figure 26.6).

The first is an **Action** (⚙) menu that affords yet another means to start recording or looping, and to control the visibility of the Detail and Extended Detail panes, and the Library window. There is a submenu for selecting an instrument to add to the document. The **Spin Monitor** item is a toggle; when it's checked, Instruments will automatically add a Spin Monitor instrument to the document whenever an application being traced hangs.

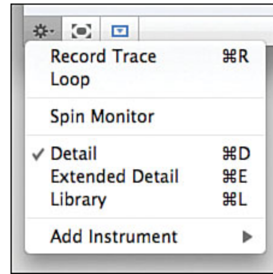

FIGURE 26.6 The controls at the lower left corner of a trace document window, with the **Action** menu displayed.

The second is the Full Screen toggle. Instruments' extensive display eats up a lot of screen area, and when you're concentrating on your analysis, you want the display to be as big as possible. Clicking this button fills the screen with the contents of the window. Click it again to return to normal windowing.

WARNING

In version 1.0 of Instruments, on some graphics cards, full-screen mode simply turns your screen black. To bail out, press ⌘**Q** to start a quit, and press **Return** repeatedly to accept all the document-saving sheets, until your desktop reappears.

The third button shows and hides the Detail pane.

The Library

Instruments get into a document either by being instantiated from a template, or by being dragged in from the Library window.

The Library (**Window > Library**, ⌘L) window lists all the known instruments. Initially this is a repertoire of Apple-supplied tracks, but it is possible to add your own. The main feature of the window (see Figure 26.7) is the list of all known instruments. Selecting one fills the pane below the list with a description (which for now is the same as the description in the list).

The library gathers instruments into groups; these are initially hidden, but can be seen if you select **Show Group Headers** from the **Action** (⚙) pop-up at the lower-left corner of the window. The **Action** menu also enables you to create groups of your own. The pop-up at the top of the window narrows the list down by group, and the search field at the bottom allows you to narrow the list by searching for text in the names and descriptions.

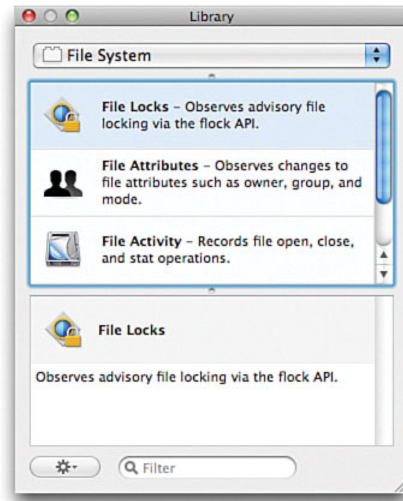

FIGURE 26.7 The Library window is dominated by the scrolling list of available instruments. The selected instrument is described in the panel below. Selecting a category from the pop-up menu narrows the list down by task, and the search field at the bottom allows you to find an instrument from its name or description.

Running an Instrument

To use an instrument, you follow three steps: configuration, recording, and, optionally, saving the results.

Instrument Configuration

Configuration inspectors vary by instrument, but some elements are used frequently.

There is a **Target** pop-up that initially points to the document's default target (set with the **Default Target** menu in the toolbar). If no default target has been selected, or if the **Default Target** menu has been set to **Instrument Specific**, the instrument's **Target** is active. You can select from processes already running, applications that Instruments had sampled before, a new application or tool of your choice, or, with many instruments, the system as a whole.

The ability to set a target for each instrument is important: It allows you to examine the behavior of an application *and* other processes with which it communicates, simultaneously.

In the **Track Display** section, there are three controls: A **Style** pop-up, a **Type** pop-up, and a **Zoom** slider.

The usual **Style** menu selects among graphing styles for the numeric data the instrument records. These may include the following:

- **Point**. Each datum is displayed as a discrete symbol in the track. You can choose the symbols in the list of available series in the inspector.
- **Line**. The track is displayed as a colored line connecting each datum in the series. You can choose the color in the list of the available series.
- **Filled Line** is the same as **Line**, but the area under the line is colored.

- **Peak** shows the data collected by an instrument that records events (such as the Core Data instruments) as a vertical line at each event. Every time something happens, the trace shows a blip.
- **Block** is a bar graph, showing each datum as a colored rectangle. In instruments that record events, the block will be as wide as the time to the next event.

The **Type** menu offers two choices for instruments that can record more than one data series. **Overlay** displays all series on a single graph. The displayed data will probably overlap, but in point and line displays this probably doesn't matter, and filled displays are drawn translucently, so the two series don't obscure each other. **Stacked** displays each series in separate strips, one above the other.

NOTE

Most instruments record events, not quantities that vary over time. In fact, the data displayed may not even be a continuous variable, but may be a mere tag, like the ID of a thread or a file descriptor. The Peak style is the most suitable style for event recordings. Such displays are still useful, however, because they give you a landmark for examining the matching data in the other tracks.

Zoom increases the height of the instrument's track. This is especially handy in stacked displays, enabling you to view multiple traces without squishing them into illegibility. The slider clicks to integer multiples of the standard track height, from 1 to 10 units.

You can change the **Track Display** settings even after the instrument has collected its data. You can find a shortcut for the **Zoom** slider in the **View** menu, as **Increase Deck Size** (⌘+) and **Decrease Deck Size** (⌘–).

One disadvantage of the inspector system is that inspectors are of fixed size, and can be quite tall. If a track is low on the screen (which it may have to be, if it is low in a multitrack document), it might run off the bottom, obscuring the **Done** button that dismisses the inspector. Fortunately, you can also dismiss an inspector by pressing the **i** button again. The only workaround that allows you to get at the options at the bottom of the inspector is to drag the track to the top of the document, make the setting, and, if you want, drag it back.

Recording

There is more than one way to start recording in Instruments.

The most obvious is to create a trace document and click the **Record** button in the toolbar. Recording starts, you switch to the target application, perform your test, switch back to Instruments, and click the same button, now labeled **Stop**.

The first time you record into a document that contains a User Interface instrument, the recording button will be labeled **Record**, as usual. Once the UI track contains events, the recording button is labeled **Drive & Record**. When you click it, no new events are not recorded into the UI track; instead, the events already there are *replayed* so you can reproduce your tests.

If you want to record a fresh User Interface track, open the configuration inspector (with the **i** button in the instrument's label) and select **Capture** from the **Action** pop-up. The recording button will revert to **Record**.

A second way to record is through the Quick Start feature, which allows you to start recording with a systemwide hotkey combination. To set a hotkey, open Instruments' Preferences window and select the **Quick Start** tab. This tab includes a table listing every system- and user-supplied template. Double-click in the column next to the template you choose, and press your desired hotkey combination. The combination must include at least two modifier keys (such as Command, Shift, and so on).

With the hotkey set, move the cursor over a window belonging to the application you want to target, and then press the key combination. Instruments will launch if it is not running already, open a new trace document behind your application with the template you selected, target it on your application, and start recording. To stop, make sure your cursor is over one of the target's windows, and press the key combination again (or switch to Instruments and click **Stop**).

The requirement to point the cursor at one of the target's windows allows you to run simultaneous traces on more than one application.

To remove a hotkey, select the template in the **Quick Start** table, and press the **Delete** key.

The third way to record is through the Mini Instruments window. Selecting **View** > **Mini Instruments**, or clicking the **Mini** button in the toolbar of any document window, hides all of Instruments' windows and substitutes a floating heads-up window listing all of the open trace documents (see Figure 26.8).

The window lists all of the trace documents that were open when you switched to Mini mode; scroll through by clicking the up or down arrowheads above and below the list. At the left of each item is a button for starting (round icon) or stopping (square icon) recording, and a clock to show how long recording has been going on. Stopping and restarting a recording adds a new run to the document.

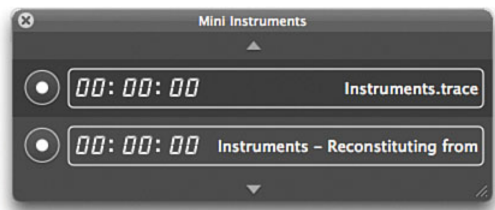

FIGURE 26.8 The Mini Instruments heads-up window. It lists each open trace document next to a clock and a recording button. Scroll through the list using the arrowheads at top and bottom.

As with Quick Start keys, Mini Instruments has the advantages that it's convenient to start recording in the middle of an application's run (handy if you are recording a User Interface track that you want to loop) and that you can control recording without switching out of the target application (which can also impair a UI recording).

You return to the full display of Instruments by clicking the close (**X**) button in the upper-left corner of the Mini Instruments window.

Saving and Reopening

As with any other Macintosh document, you can save a trace document. The document will contain its instruments and all the data they've collected. There can be a lot of data, so expect a trace document to be large—on the order of tens of megabytes.

It's likely that you will come to need a uniform layout of instruments that isn't included in the default templates provided by Apple. You can easily create templates of your own, which will appear in the template sheet presented when you create a new trace document. Configure a document as you want it, and select **File** > **Save as Template...**.

The ensuing save-file sheet is the standard one, focused on the directory in which Instruments looks for your templates, `~/Library/Application Support/Instruments/Templates`. The name you give your file will be the label shown in the template-choice sheet. At the lower left of the sheet is a well into which you can drag an icon (for instance, if your template is for testing your application, you'd want to drop your application's icon file here), or you can click and hold the mouse button over the well to choose Apple-provided icons from a pop-up. The panel provides a text area for the description to be shown in the template-choice sheet.

- The document's suite of instruments, and their configurations, will be saved in the template.
- The template will include the default and instrument-specific targets you set.
- If you include a prerecorded User Interface track, the contents will be saved. This way you can produce uniform test documents simply by creating a new trace document and selecting the template.

As you'd expect, you can reopen a trace document by double-clicking it in the Finder, or through **File** > **Open...**. All the data is as it was when the document was saved. Clicking **Record** adds a new run to the document.

The Instruments

Here are the instruments built in to Instruments as of the time of this writing, grouped as they are in the Library window (select **Show Group Banners** from the **Action** (⚙) pop-up at bottom left).

> **NOTE**
>
> Apple is free to add or remove built-in instruments, or to change their capabilities significantly. This can't be a definitive list. For the latest information, search for the *Instruments User Guide* in the Developer Tools Reference in the Xcode Documentation window.

Most instruments are DTrace based (see the section on "Custom Instruments" later in this chapter). DTrace automatically records thread ID and a stack trace, and implicitly the stack depth, at the time of the event. Every numeric-valued property of the event is eligible for graphing in the instrument's track, which accounts for the odd offer of "Thread ID" for plotting in such instruments.

All instruments can target any single process, or all processes on the system, unless the description says otherwise.

Core Data

Core Data Saves

At each save operation in Core Data, the Core Data Saves instrument records the thread ID, stack trace, and how long the save took.

Core Data Fetches

This instrument captures the thread ID and stack trace of every fetch operation under Core Data, along with the number of objects fetched and how long it took to complete the fetch.

Core Data Faults

Core Data objects can be expensive both in terms of memory and of the time it takes to load them into memory. Often, an `NSManagedObject` or a to-many relationship is given to you as a *fault*, a kind of IOU that will be paid off in actual data when you reference data in the object.

This instrument captures every firing (payoff) of an object or relationship fault. It can display the thread ID and stack depth of the fault, as well as how long it took to satisfy object and relationship faults.

Core Data Cache Misses

A faulted Core Data object may already be in memory; it may be held in its `NSPersistentStoreCoordinator`'s cache. If you fire a fault on an object that *isn't* in the cache (a "cache miss"), however, you've come into an expensive operation, because the object has to be freshly read from the database. You want to minimize the effect of cache faults by preloading the objects when it doesn't impair user experience.

This instrument shows where cache misses happen. It records the thread ID and stack trace of each miss, and how much time was taken up satisfying the miss, for objects and relationships.

File System

These instruments record POSIX calls that affect the properties of files and directories. This does not include reads and writes; for those, see the Reads / Writes instrument under Input / Output.

File Locks

This is an event instrument that records the thread ID, stack trace, function, option flags, and path for every call to the `flock` system function.

File Attributes

For every event of changing the owner, group, or access mode of a file (`chown`, `chgrp`, `chmod`), this instrument records thread ID, a stack trace, the called function, the file descriptor number, the group and user IDs, the mode flags, and the path to the file affected.

File Activity

This is an event instrument that records every call to `open`, `close`, `fstat`, `open$UNIX2003`, and `close$UNIX2003`. It captures thread ID, call stack, the call, the file descriptor, and path.

Directory I/O

This instrument records every event of system calls affecting directories, such as creation, moving, mounting, unmounting, renaming, and linking. The data include thread ID, stack trace, call, path to the file directory affected, and the destination path.

Garbage Collection

GC Total

GC Total collects statistics on the state of garbage collection in a process (or in all garbage-collected processes) at the time collection ends. In addition to thread ID and stack traces, it records the number of objects, and bytes, just reclaimed, the number of bytes still in use, and the total number of reclaimed and in-use bytes.

Garbage Collection

This is slightly different from the GC Total instrument. It measures across the beginning and end of the scavenge phase of garbage collection. It records whether the reclamation was generational, and how long scavenging took. It also records the number of objects and bytes reclaimed.

Graphics

OpenGL Driver

This instrument taps the OpenGL drivers for the graphics displays to collect a huge number of statistics on OpenGL usage, by the target process (or the entire system), at an interval of your choosing (initially one second). The graphical trace itself doesn't signify anything, and can't be usefully configured in the inspector. The substance of the recording is to be found in the Detail table, and the Detail-control view has check boxes that determine which statistics appear there (there are nearly 60).

Input / Output

Reads / Writes

The events recorded by this instrument include `reads` and `writes` to file descriptors. Each event includes the thread ID, the name of the function being called, a stack trace, the descriptor and path of the file, and the number of bytes read or written.

Master Track

User Interface

This track records your mouse movements, clicks, and keystrokes as you work with an application. Each event carries a thumbnail of the screen surrounding the mouse cursor.

The UI track's events serve as landmarks for the internal program events recorded by other instruments, but the real utility—the reason this is called a *master track*—is that once a UI track is recorded, it can be played back; it is said to "drive" the application. When a UI track containing events is available, the **Record** button is relabeled **Drive & Record**, and clicking it will replay the human-interface events.

You can divert from driving by using the **i** button in the instrument's label to open the instrument's configuration inspector, and switching the **Action** pop-up from **Drive** to **Capture**.

For an extended example of using the User Interface track, see the "Human-Interface Logging" section of Chapter 19.

Memory

Shared Memory

The Shared Memory instrument records an event when shared memory is opened or unlinked. The event includes calling thread ID and executable, stack trace, function (`shm_open`/`shm_unlink`), and parameters (name of the shared memory object, flags, and `mode_t`). Selecting an event in the Detail table puts a stack trace into the Extended Detail pane.

ObjectAlloc

We saw ObjectAlloc and Leaks in Chapter 19, when we debugged a memory leak in `Linear`.

ObjectAlloc collects a comprehensive history of every block of memory allocated during the run of its target. It can track the total number of objects and bytes currently allocated in an application because it records every allocation and deallocation, and balances them for every block's address.

The main Detail Table view lists every class of block that was allocated, and aggregate object and byte counts; use the **Inspection Range** tool to focus on allocations and deallocations within a given period. The classes can be checked to plot them separately in the trace.

Mousing over a classname reveals an arrow button; if you click it, the Detail table drills in to a table of every block of that class allocated in the selected time interval. Drilling in on the address field in one of these reveals a history of every event that affected that address—`malloc`s and `free`s at least, and if **Record Reference Counts** was checked in the configuration inspector before launching, reference-counting events as well. Mac OS X may use the same address more than once as memory is recycled; you'll usually see `malloc` events after every `free` but the last one.

The breadcrumb control below the Detail pane reflects each stage in the drilling-down process. Click the label for an earlier stage to return to it.

The track-style options in the configuration inspector include **Current Bytes**, a filled-line chart that shows the total current allocations; **Stack Depth**, a filled-line chart that shows how deep the call stack is at each allocation event; and **Allocation Density**, a peak graph showing the change in allocated bytes at each event (essentially a first derivative of the Current Bytes display).

In the Outline view, the top level lists the allocation classes. Below them are stack trees for all the allocations of those classes. The data-mining and Extended Detail tools are available in this view.

The Diagram view of the Detail table lists every allocation event. As in the Detail view, clicking the arrow button in an address view displays a history of allocation, deallocation, and reference-count events for that address.

ObjectAlloc can be run only against a process that Instruments launched, and you should pay attention to the **Launch Configuration** switches in the configuration inspector before recording.

The ObjectAlloc instrument is powerful and subtle. It merits an entire section in the *Instruments User Guide*. Search for "Analyzing Data with the ObjectAlloc Instrument" in the Developer Tools Reference in the Xcode Documentation window.

Leaks

Leaks also tracks the allocation and deallocation of objects in an application (which must be launched by Instruments itself), but does so to detect the objects' being allocated and then lost—in other words, memory leaks. Leaks does not rely just on balancing allocations and deallocations; it periodically sweeps your program's heap to detect blocks that are not referenced by active memory.

The table view of the Detail pane lists every object that was allocated in the selected time interval, but found to have no references at the end. The line items show the percentage of total leakage the block represents, its size, address, and class. Selecting a line fills the Extended Detail pane with a general description and a stack trace of the allocation. Each address entry has an arrow button that drills down to the allocation, deallocation, and reference-count events for that address. You have the entire history of the block; you should be able to determine where an over-retain occurred. Reducing the inspection range on the trace will not narrow this list; it's for the entire history of the address.

The stack tree in the outline view goes from the `start` function in the runtime down the various paths to the allocating function, usually `calloc` in the case of Objective-C objects. Paring system libraries from the tree will quickly narrow the list down to the calls in your code responsible for creating leaked blocks.

The configuration inspector for the Leaks instrument controls how the trace is displayed, but the actual behavior is controlled by the control section of the Detail pane. The defaults are useful, but expose the Detail pane before you run to verify the settings are

what you want. The settings control whether memory sweeps for unreferenced blocks are to be performed, whether the contents of leaked blocks will be retained for inspection, and how often to perform sweeps.

System

Activity Monitor

This instrument is too varied to explain fully here, but its features should be easy to understand if you explore its configuration inspector. It collects 31 summary statistics on a running process, including thread counts, physical memory usage, virtual memory activity, network usage, disk operations, and percentages of CPU load. This instrument more or less replaces BigTop as a graphical presentation of application activity.

Remember that you can have more than one Activity Monitor instrument running, targeting different applications or the system as a whole.

The Detail table lists the statistics for every process covered by the instrument. Moving the playback head makes the table reflect the processes and statistics as of the selected time. The hierarchical view arranges the processes in a parent-and-child tree.

Sampler

Sampler is the poor man's Shark. It samples the target application at fixed intervals (10ms by default, but you can set it in the inspector), and records a stack trace each time. It does not record the position in the target down to the instruction, and the analysis tools are limited, but it's often good enough to find bottlenecks or determine where an application has hung.

Sampler was formerly supplied as a standalone application. The Sampler application supplied with Xcode 3 simply opens the CPU Sampler template in Instruments.

Sampler must have a specific process or launched application as its target; sampling the entire system makes no sense.

Spin Monitor

Spin Monitor is the Instruments version of the Spin Control application. To mimic Spin Control, set the target to **All Processes** and leave the trace document recording. Whenever an application (or the target application) shows the spinning-rainbow cursor, indicating it has stopped accepting human-interface events, the Spin Monitor becomes Sampler, building a stack tree while the spin continues.

The table view of the Detail panel has a top-level entry for each spinning incident. Within these are items for each sample in the incident, which expand to show each thread in the target. The outline view displays an aggregate tree of stack traces for all the samples in each incident; the Call Tree controls in the Detail panel become available.

Process

For each start (`execve`) and end (`exit`) event in a process, this instrument records thread ID, stack trace, process ID, exit status, and executable path.

Network Activity Monitor

This is actually the Activity Monitor with four of eight network statistics active: Network Packets/Bytes In/Out Per Second. It omits the absolute numbers of packets and bytes transmitted.

Memory Monitor

This is the Activity Monitor with Physical Memory Used/Free, Virtual Memory Size, and Page Ins/Outs checked.

Disk Monitor

This is the Activity Monitor with Disk Read/Write Operations Per Second, and Disk Bytes Read/Written Per second checked.

CPU Monitor

This is the Activity Monitor with % Total Load, % User Load, and % System Load selected.

Threads/Locks

JavaThread

The JavaThread instrument is unique, in that it does not display its trace as a vertical graph. Instead, the trace is a stack of bars, extending horizontally through time, that represent the threads in a Java application. A bar appears when a thread starts. It is colored green while it runs, yellow while it waits, and red while it is blocked. The bar disappears when the thread halts. A sample is taken whenever such thread events occur; the Detail table shows the time of day at which the sample was taken, and the number of threads existing at that time.

Clicking the arrow button in the clock time of an item drills down to the details of the event: a table listing all threads by name, their priorities, states, number of monitors, and whether they are daemon threads. The Extended Detail view for a thread shows a stack trace, and a list of monitors the thread owns.

User Interface

Cocoa Events

Cocoa Events records an event at every call to `-[NSApplication sendEvent:]`. It captures the thread ID, stack trace, the event code, and a string (such as "Left Mouse Down") that characterizes the event.

Carbon Events

Carbon Events records an event at every return from `WaitNextEvent`. It captures the thread ID, stack trace, the event code, and a string (such as "Key Down") that characterizes the event.

Custom Instruments

Some of the instruments included in Instruments consist of code specially written for the task. Most involve no code at all. They are made from editable templates. You can examine these instruments yourself—which may be the only way to get authoritative details on what an instrument does—and you can create instruments of your own.

Let's see what a scripted instrument looks like. Create a trace document from the File Activity template, select the Reads / Writes instrument, and then **Instrument > Edit 'Reads/Writes' Instrument...** (or simply double-click the instrument's label). An editing sheet (see Figure 26.9) will appear, with fields for the instrument's name, category, and description, and a long scrolling list *of probes*, handlers for events the instrument is meant to capture.

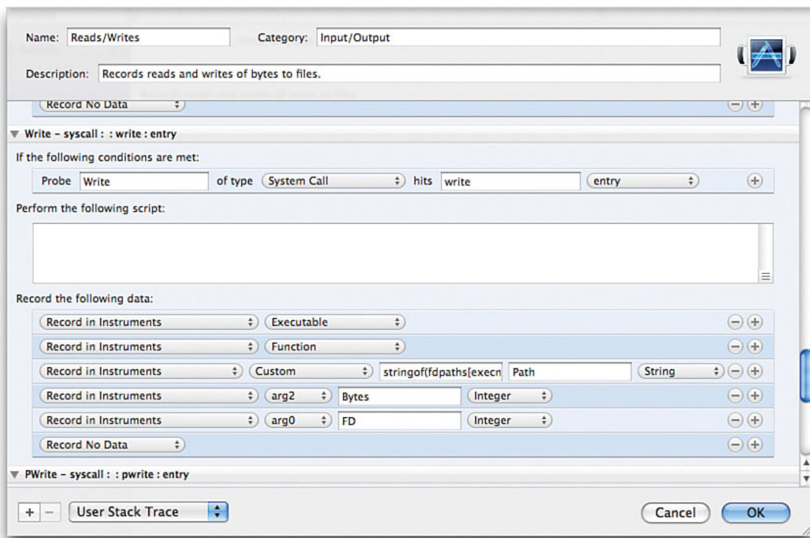

FIGURE 26.9 The Edit Instrument sheet for the Reads / Writes instrument. The sheet is dominated by an editable list of events the instrument is to capture. The portion that specifies how to record entries to the system `write` function is shown here.

Figure 26.9 shows the event list scrolled to the condition called Write, in the domain System Call, for the symbol `write`. It is to trigger when `write` is entered. Next comes the text of a script to be executed when the probe is triggered. Instruments uses the *DTrace* kernel facility, which has its own scripting language; for instance, this event might put the time at which the event occurred into an instance variable of the probe, so that a `write`-exit probe could calculate the duration of the call and record it. In this case, the scripting text is blank.

Then comes a series of items specifying what information is to be kept, for the trace graph or for the Detail view. In the case of Reads / Writes, this is

- The name of the executable
- The name of the function
- The first argument (the file descriptor), which is an integer to be labeled **`FD`**
- The third argument (the size of the write), which is an integer to be labeled **`Byte`**
- A string, to be labeled **`Path`**, calculated from an expression in the Instruments scripting language: A path, derived from the file descriptor within the executable.

Integer-valued records are included in the configuration inspector's list of **Statistics to Graph**, and are eligible to display in the instrument's trace. This accounts for the odd presence of Thread ID (which is automatically captured in every case) in the list of available plots. By default, the Stack Depth statistic is selected.

The customization sheet is a front end for the scripting language for the kernel-provided DTrace service; only kernel-level code is capable of detecting call events in every process. The section "Creating Custom Instruments with DTrace," in the *Instruments User Guide*, offers enough of an introduction to the language to get you started on your own instruments.

To make your own instrument, start with **Instrument** > **Build New Instrument...** (⌘**B**). An instrument-editing sheet will drop from the front trace document, and you can proceed from there.

26

If you become a DTrace expert, you might find it more convenient, or more flexible, to write your scripts directly, without going through the customization sheet. Select **File** > **DTrace Script Export...** to save a script covering every instrument in the current document, and **File** > **DTrace Data Import...** to load a custom script in. You can export DTrace scripts only from documents that contain DTrace instruments exclusively.

The stack trace in the Extended Detail view provides another way to create a custom instrument. Select one of the function frames in the listing and then **Trace Call Duration** from the stack trace's **Action** (⚙) menu. Instruments will add a custom instrument to the current document that triggers on entry and exit, to record how long it took to execute the function.

WARNING

DTrace is a new feature of Mac OS X 10.5, and it executes as part the operating system kernel. That means the entire system is vulnerable to a crash (a "kernel panic") if something goes wrong. This should be rare, and should get rarer, but I've had it happen with Instruments 1.0 on Mac OS X 10.5.2. Make sure your documents are all saved, and back your system up frequently.

The Templates

When you create a new Instruments document, a sheet drops down offering a choice among templates, preconfigured sets of instruments for common tasks (see Figure 26.10). Click the configuration of your choice, and then click one of the buttons at the bottom of the sheet:

- **Open an Existing File...** abandons the new document and presents a standard file-selection dialog for opening an old one. This is simply a convenience, effectively canceling the sheet and performing **File > Open...**.
- **Cancel** closes the untitled document without saving.
- **Choose** creates the document and populates it with the instruments for the selected template.
- **Record** does a lot of work. It creates the document and populates it with the selected template's instruments. Then it starts recording. When a trace document doesn't have a default target application—as a newly instantiated document would not—Instruments has to associate an application with each instrument in the document. A variant on the standard open-file dialog appears, enabling you to pick the target application or tool, and to specify arguments and environment variables. You can specify one target for all the instruments by checking the **Apply to All Instruments** box. When all the targets are set, Instruments launches them and starts recording.

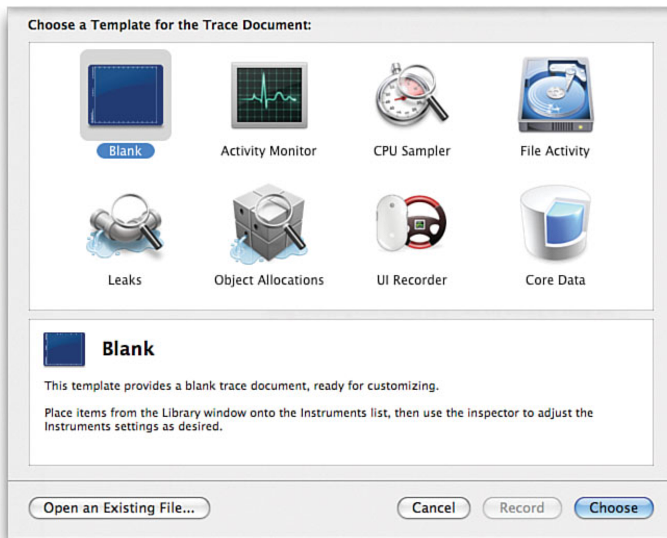

FIGURE 26.10 The choose-template sheet that introduces each new trace document in Instruments. There are eight standard templates, including a blank template that has no instruments in it at all. Selecting a template fills the lower view with a brief description of the template.

These are the standard templates Apple supplies. As mentioned in "Saving and Reopening," you can create templates of your own by composing a trace document to your needs, and using **File > Save as Template...** to save it.

- **Blank** contains no instruments at all. You add the ones you want by dragging them in from the Library.
- **Activity Monitor** contains the Activity Monitor instrument. It's a comprehensive instrument, and this template can be thought of as a recordable version of the Activity Monitor application, or as a nicer version of BigTop.
- **CPU Sampler** gives you the Sampler and CPU Monitor instruments. The one provides a statistical by-function profile of the target application, and the other, the CPU load at the same times. The Sampler application now launches Instruments and opens this template.
- **File Activity** sets you up with File Activity, Reads/Writes, File Attributes, and Directory I/O.
- **Leaks** is the template we used in Chapter 19. It provides the ObjectAlloc and Leaks instruments to track the rate of object creation, and to verify that what you allocate, you also free. If you are interested in what your application is doing with memory, this is the template to use, rather than Object Allocations, which provides only ObjectAlloc.
- **Object Allocations** contains only the ObjectAlloc instrument. It is intended as a substitute for the ObjectAlloc application from earlier editions of the developer tools, and in fact running the current version of the ObjectAlloc application simply runs Instruments and instantiates this template.
- **UI Recorder** provides the User Interface instrument only. This template can be used to construct lifetime or looping scripts to verify the correct operation of a program, or you can add tracks to capture data as events occur.
- **Core Data** includes Core Data Fetches, Core Data Cache Misses, and Core Data Saves. It does not include Core Data Faults (you could drag it in), but it hits all the events that really impact the performance of a Core Data application.

Summary

Instruments is a big topic, and we covered much of it. We started with a tour of the trace document window, and moved on to populating it from the Library window. We covered general principles of how to configure an instrument track.

We saw the various ways to start and stop recordings, including human-interface recordings that can be played back to generate repeatable tests for your applications.

We took inventory of the instruments and document templates Apple supplies, and how to create your own.

As your needs and expertise progress, you'll want to consult the *Instruments User Guide*, which you can find in the Xcode Documentation browser.

Index

Symbols

^D (end-of-file character), 34
#pragma mark lines, 57
%%%{PBX}%%% substitution markers, 227-228
64-bit applications, 274
64-bit architectures, builds for, 363

A

accessors
for attributes, 56
automatically generating, 50
redesigning data model, 250-251
ACTION build variable, 478
Action menu (trace document window), 445
action methods, 85
Action template, 486-487
actions
defined, 76, 486
outlets and, 84-85
Activate button (Debugger window), 31
active targets, 141, 148
Activity Monitor instrument, 454, 459
ADC (Apple Developer Connection), 3, 503
Address Book Action Plug-In for C project, 493
Address Book Action Plug-In for Objective-C project, 493
Advanced Mac OS X Programming (Dalrymple and Hillegass), 501
Aggregate target, 499
all-in-one layout for projects, 350-351
allocation calls, checking for memory leaks, 318-319
AllowFullAccess key, 164
AllowInternetPlugins key, 164
AllowJava key, 164

AllowNetworkAccess key, 165
AllowSystem key, 165
analysis results, viewing in Shark, 423-425
antecedents. *See* dependencies
API documentation, creating, 238-241
Apple Developer Connection (ADC), 3, 503
AppleGlot glossaries, 313
AppleScript Application project, 487
AppleScript Automator Action project, 486
AppleScript Document-Based Application project, 487
AppleScript Droplet project, 487
AppleScript Xcode Plug-In project, 493
Applet legacy target, 500
Applet target (Java), 498
application bundles, 156-158
  Info.plist file, contents of, 158-165
  installing frameworks in, 217, 219
Application legacy target, 499-500
Application Properties Info window, 88-91
Application target (Carbon), 496
Application target (Cocoa), 497
Application target (Java), 498
Application template, 487-489
applications. *See also* Cocoa applications
  attaching Debugger window to, 376
  changing language preferences, 310
  Info.plist keys for, 160-163
  roles, 91
  running, 31
architectures, testing for multiple, 179-180
archives, unpacking, 404-405
archiving custom views, 185
ARCHS build variable, 481
Arguments tab (Executable Info window), 388
arranged objects, 84
arrows, 69
  in file-comparison window, 108
ASCII property lists, 132-133
assembly code, 40
assigning files to framework targets, 205-210
associated breakpoints, 376
associating
  data files with Cocoa applications, 88-91
  file types with text editors, 468-469
  files with directories, troubleshooting, 113-114
  NIB files with projects, 183
  projects with repository, 104
  UTIs with data files, 284
ATSApplicationFontsPath key, 163
attaching Debugger window to applications, 376
attributes
  accessor methods for, 56
  in data model files, setting, 247
Attributes Inspector, 82
Audio Unit Effect project, 489
Audio Unit Effect with Carbon View project, 489
Audio Unit Effect with Cocoa View project, 489
Audio Unit Instrument project, 489
Audio Units template, 489
auto-complete in Code Sense, 125
auto-properties, enabling in Subversion, 98
automating docset build process, 233-235
Autosizing section (Size inspector), 69

B

background tasks, viewing, 332
backward compatibility. *See* cross-development
base version of files, latest version versus, 108
BBEdit, 128, 133, 505
***Beginning Mac OS X Programming* (Trent and McCormack), 502**
***Beginning Xcode (Programmer to Programmer)* (Bucanek), 501**
BigTop, 432
binaries, universal, 274-276
  creating in example build transcript, 365
  Intel-porting issues in Linear example, 276
  testing, 276-277

binary format (Core Data), 256
binary property lists, 133
bindings, 78
advantages of, 81
creating, 80-82
in Interface Builder for created entities, 263
value binding, 82-83
Bindings Inspector, 80-82
Blank template (Instruments application), 459
blocks
checking for memory leaks, 317-318
visualization of, 335-336
blogs for additional information, 504
blue guidelines in Interface Builder, 66
books for additional information, 501-502
Boolean text searches in documentation, 471
bottom-up view (Shark analysis results), 423-424
branches directory, 100
branching (in version control), 243-245
breakpoint commands, 381-383
breakpoint conditions, 383-384
breakpoints
associated breakpoints, 376
enabling/disabling, 30, 390-391
grouping, 376
lazy symbol loading and, 384-385
removing orphan, 376
setting, 30-31, 373-374
Breakpoints button (Debugger window), 32
Breakpoints menu (Debugger window), 32
Breakpoints window, 32, 382
setting breakpoints, 373-374
broken links, Interface Builder and, 461-462
browsing, enabling in Documentation window, 235-237
BSD Dynamic Library project, 490
BSD Static Library project, 495
BSD target template, 496
bug reports, Web site for, 503
Build and Go button (Debugger window), 31
build configurations, 92, 368-370
configuration files for, 370
Debug versus Release, 275
Release build configuration, 323-325
stripping and, 325
targets versus, 92
build dependencies, 403. *See also* dependencies
build directory, setting preferences for single, 465
build errors, 23-26
Build Java Resources build phase, 343
build phases
adding to targets, 342
in example build transcript, 360-364
files in, 354
list of, 342-343
Run Script build phase, 357-358
for targets, 41
build process
distributed builds, 397-399
for docsets, automating, 233-235
Build Resource Manager Resources build phase, 343
Build Results window, 23-24, 176, 349, 359
build rules, creating custom, 356-357
build sets, defined, 398
build settings, viewing list of, 466
build system. *See also* build configurations; build variables; cross-development; example build transcript
build settings, viewing list of, 466
custom build rules, creating, 356-357
explained, 353-355
Run Script build phase, 357-358
settings hierarchy, 366-368
xcodebuild tool, 365-366
Build tab (Target Info window), 177
build targets, list of, 478-479
build transcript, 24. *See also* example build transcript

How can we make this index more useful? Email us at indexes@samspublishing.com

build variables, 355-356
- build targets, list of, 478-479
- bundle locations, list of, 480
- compiler settings, list of, 481-482
- deployment variables, list of, 482-483
- destination locations, list of, 479-480
- environment variables, list of, 477-478
- Run Script build phase and, 358
- search paths, list of, 482
- source locations, list of, 479
- source trees, 483
- viewing, 475-476

building. *See also* build process; build system; external build system projects
- Core Data data files, 258-259
- interfaces for created entities, 260, 262-264
- projects
  - cleaning before, 219
  - in Organizer window, 409-411
  - saving first, 23
- sample application, 91
- targets, 15
  - cleaning before, 268

built-in accessors, 50
BUILT_PRODUCTS_DIR build variable, 479
Bumgarner, Bill, 504
Bundle legacy target, 499
bundle locations, list of, 480
Bundle template, 489
bundles, 156, 168. *See also* application bundles; packages
- Copy Bundle Resources build phase in example build transcript, 361
- defined, 62
- docsets as, 230
- load-path references in, 218
- types of, 156

buttons
- adding to application windows, 66-67
- connecting to NIB files, 84-85
- in Debugger window, 31-32

C

C++ Dynamic Library project, 491
C++ Standard Dynamic Library project, 491
C++ Tool project, 490
calculating linear regression, 20-22
call trees, filtering, 444
call-tree view
- Leaks instrument, 318-319
- ObjectAlloc instrument, 320

callstack data mining in Shark, 425-427
Carbon Application project, 487
Carbon Bundle project, 490
Carbon C++ Application project, 487
Carbon C++ Standard Application project, 487
Carbon Dynamic Library project, 491
Carbon Events instrument, 455
Carbon Framework project, 491
Carbon Static Library project, 495
Carbon target template, 496-497
carbon-dev mailing list, 503
case sensitivity of filenames, 462
categories
- adding to classes, 118-121
- naming conventions, 122

CFAppleHelpAnchor key, 160
CFBundleAllowMixedLocalizations key, 162
CFBundleDevelopmentRegion key, 159
CFBundleDisplayName key, 161, 308
CFBundleDocumentTypes key, 160
CFBundleExecutable key, 160
CFBundleGetInfoString key, 159
CFBundleHelpBookFolder key, 161
CFBundleHelpBookName key, 161
CFBundleIconFile key, 159
CFBundleIdentifier key, 159
CFBundleInfoDictionaryVersion key, 159
CFBundleLocalizations key, 163
CFBundleName key, 161
CFBundlePackageType key, 159
CFBundleShortVersionString key, 159
CFBundleSignature key, 159
CFBundleURLTypes key, 160

CFBundleVersion key, 159
CFPlugIn Bundle project, 490
CFPlugInDynamicRegisterFunction key, 163
CFPlugInDynamicRegistration key, 163
CFPlugInFactories key, 163
CFPlugInTypes key, 163
CFPlugInUnloadFunction key, 163
CFZombieLevel environment variable, 389
checking out working copies of projects, 103-104
CHUDRemover, 436
Class Actions section (Identity Inspector), 76
Class Browser window, 345-346
class modeler tool, 346-348
classes
- adding categories to, 118-121
- custom classes, adding to NIB files, 76
- document classes, 91
- initialize method, 191
- principal class for applications, 90
- test classes, creating, 168
- view classes, adding, 183

cleaning
- projects, 219
- targets before building, 268

clearing debugging log, 374
CloseboxInsetX key, 164
CloseboxInsetY key, 164
Cocoa Application project, 487
Cocoa applications
- associating data files with, 88-91
- converting to Objective-C 2.0, 172-173
- embedding tools in, 52-54
- MVC design pattern, 48, 54-62
- property list data types, 118
- sample application
  - building, 91
  - controller object, 51, 75-92
  - model object, 48-50
  - tasks of, 47
  - view object, 51, 63-73
- starting new projects, 52

Cocoa Automator Action project, 486
Cocoa Bundler project, 490
***Cocoa Design Patterns* (Buck), 501**
Cocoa Document-Based Application project, 487
Cocoa Dynamic Library project, 491
Cocoa Events instrument, 455
Cocoa Framework project, 491
Cocoa legacy target, 500
***Cocoa Programming for Mac OS X, Third Edition* (Hillegass), 502**
Cocoa Simulator, 69
Cocoa Static Library project, 495
Cocoa target template, 497-498
cocoa-dev mailing list, 503
Cocoa-Python Application project, 488
Cocoa-Python Core Data Application project, 488
Cocoa-Python Core Data Document-Based Application project, 488
Cocoa-Python Document-Based Application project, 488
Cocoa-Ruby Application project, 488
Cocoa-Ruby Core Data Application project, 488
Cocoa-Ruby Core Data Document-Based Application project, 488
Cocoa-Ruby Document-Based Application project, 488
CocoaBuilder Web site, 504
CocoaCheerleaders Web site, 504
CocoaDev wiki, 504
CocoaDevCentral Web site, 504
CocoaHeads Web site, 504
code completion, 332-333
Code Focus, 335-336
Code Sense, 240, 332-333
- auto-complete in, 125
- class modeling and, 347
- in external build system projects, 416-417

code. *See* source code
CodeWarrior targets, 12
colors, configuring from property lists, 189-192

columns, adding to Groups & Files list, 103
command-line terminal. *See* Terminal application
command-line tools
creating as projects, 14
linear regression, 22-23
debugging mode setup, 25-26
illegal operations in, 29-30
Command-Line Utility template, 490
commands
breakpoint commands, 381-383
from human interface, responding to, 51
committing project files in version control, 110
comparing project files in version control, 107
compatibility. *See* cross-development
Compile AppleScripts build phase, 342
Compile Sources build phase, 41, 342, 361-364
compiled code, order of operations in, 41
compiler settings, list of, 481-482
compilers, defined, 39
compiling
data models in example build transcript, 364
projects
distributed builds, 397-399
precompiled headers, 395-396
predictive compilation, 396
source code, 39-41
XIB files in example build transcript, 360
completion prefixes, 139
Concurrent Versions System (CVS), 94, 104
condensed layout for projects, 351-352
conditions, breakpoint, 383-384
CONFIGURATION build variable, 478
configuration directories
creating, 96
editing, 96-98
configuration files for build settings, 370
configurations. *See* build configurations
configuring
colors from property lists, 189-192
Groups & Files list, 345
instruments, 446-447
projects in Organizer window, 409-411
conflict markers, 110
conflict resolution in version control, 110-113
connecting
buttons to NIB files, 84-85
outlets and objects, 184-187
Console button (Debugger window), 32
Console window, 32, 349-350
clearing debugging log, 374
opening, 15
writing standard error stream to, 377-379
content type in metadata listings, 283-284
Contents directory (application bundles), elements in, 156-158
contents
of Info.plist file, viewing, 158-165
of repository, viewing, 103
CONTENTS_FOLDER_PATH build variable, 480
Continue button (Debugger window), 32
controller classes, editing for data modeling, 256-258
controller objects (MVC design pattern) for sample application, 48, 51, 75-83, 85-92
Controller phase (MVC design pattern)
creating custom views, 181-183
writing property lists, 125-126
controllers
adding to NIB files, 77-79
arranged objects, 84
creating bindings, 80-82
linking outlets, 79-80
source code for sample application, 85-88
converting
Cocoa projects to Objective-C 2.0, 172-173
units of measurement, 200
Copy Bundle Resources build phase, 54, 342, 361
Copy Files build phase, 342
Copy Files Target Description target, 499
Copy Headers build phase, 342
copying
files to docsets, 231-233

structural files in example build transcript, 359-360

copyright notice

correcting in gatherheaderdoc utility, 224-225

setting default content for, 56

Core Data, 243. *See also* data modeling

building/running data files, 258-259

compiling data models in example build transcript, 364

data model files

creating, 245-247

editing, 247

redesigning, 247-256

embedding metadata in, 291, 294-297

entities, creating, 259-266

instruments, list of, 450

object storage formats, 256

relationships, creating, 260

storage types, 91

Core Data Application project, 488

Core Data Cache Misses instrument, 450

Core Data Document-Based Application project, 489

Core Data Document-Based Application with Spotlight Importer project, 489

Core Data Faults instrument, 450

Core Data Fetches instrument, 450

Core Data Saves instrument, 450

Core Data template (Instruments application), 459

Core Foundation

property list data types, 118

zombies and, 389

CoreFoundation Tool project, 490

CoreServices Tool project, 490

correlation coefficient, 20

CPU Monitor instrument, 455

CPU Sampler template (Instruments application), 459

creator codes, 89

Credits.rtf file, localization, 304-305

cross-development

with multiple SDKs, 277-279

NIB compatibility, 271-273

SDKs for, 267-270

universal binaries, 274-277

weak linking, 271

Xcode version compatibility, 279

CSResourcesFileMapped key, 160

custom build rules, creating, 356-357

custom classes, adding to NIB files, 76

custom executables, creating, 417

custom instruments, 456-457

custom metadata keys, declaring, 288-290

custom views

creating

configuring colors from property lists, 189-192

Controller phase (MVC design pattern), 181-183

delegate objects, usage of, 187-189

displaying window for, 196

drawing the view, 192-195

View phase (MVC design pattern), 183-187

debugging, 198-201

testing, 196-198

unarchiving, 185

CVS (Concurrent Versions System), 94

revision numbers, incrementing, 104

D

dashboard widgets, Info.plist keys for, 164-165

Dashcode.mpkg, 3

data design, redesigning in Core Data, 247-256

accessors, 250-251

DataPoint class, 248-249

initializers, 249-250

MyDocument class, 255-256

Regression class, 251-255

data files, associating
with Cocoa applications, 88-91
UTIs with, 284
data formatters, debugging with, 379-381
data mining callstack data in Shark, 425-427
data model files
compiling in example build transcript, 364
creating, 245-247
editing, 247
redesigning, 247-256
accessors, 250-251
DataPoint class, 248-249
initializers, 249-250
MyDocument class, 255-256
Regression class, 251-255
data modeling. See also Core Data
building/running data files, 258-259
data model files
compiling in example build transcript, 364
creating, 245-247
editing, 247
redesigning, 247-256
entities, creating, 259-266
Interface Builder, editing controller classes, 256-258
relationships, creating, 260
data points, adding to created entities, 264-265
data sources
adding to created entities, 264-265
viewing for created entities, 265-266
data types in property lists, 117-118
data validation, 302-304
DataPoint class
redesigning data model, 248-249
source file, creating, 55-58
datatips, 391-392
Deactivate button (Debugger window), 31
dead code, 325
stripping, 327-328
Debug build configuration, 368
Release build configuration versus, 216, 275, 318
Debug command, 31
debug frameworks, 374
Debugger page (all-in-one layout), 350
debugger strip in editor panes, 336-337
Debugger window, 27-28. *See also* debugging
associated breakpoints, 376
attaching to applications, 376
breakpoints
grouping, 376
removing orphan, 376
setting, 30-31, 373-374
clearing log, 374
debug frameworks, 374
DWARF and STABS formats, 376
fixing code in, 35-37
GDB log, 377
global variables, 375
in-editor debuggers versus, 392
KVO (key-value observing), 376
line-ending styles in, 377
stepping through code, 33-35
tail-recursive functions, 377
toolbar buttons, 31-32
vertical layout, 373
watchpoints, setting, 375
debugging. *See also* Debugger window; resources
additional resources for, 393
breakpoint commands, 381-383
breakpoint conditions, 383-384
custom views, 198-201
data formatters, 379-381
datatips, 391-392
external build system projects, 418
lazy symbol loading, 384-385
Mini Debugger window, 389-391
Quartz Debug, 433-435
writing standard error stream to Console, 377-379
zombies, 385-389
debugging information, system tables versus, 326
debugging mode setup for linear regression project, 25-26

declaring
custom metadata keys, 288-290
UTIs, 284-286
Dedicated Network builds, 397, 399
default content for copyright notice, setting, 56
default layout for projects, 349
Build Results window, 349
Console window, 349-350
SCM Results window, 350
default scripts in Organizer window, 413
default window layout, changing, 464
defaults command-line tool, 466
Definition Bundle project, 487
delegate objects in custom views, 187-189
deleting. *See* removing
dependencies
adding to projects, 53
build dependencies, 403
in makefiles, 353
in Xcode build system, 354
dependent targets, 148-149
dependent tests, 176-179
deployment postprocessing, 325
deployment targets, setting, 270
deployment variables, list of, 482-483
DEPLOYMENT_POSTPROCESSING build variable, 482
DERIVED_FILE_DIR build variable, 480
designable.nib file, 327
designing libraries, 143-146
destination locations, list of, 479-480
detail list, defined, 14
Detail list (ObjectAlloc instrument), 320
Detail pane (trace document window), 442-444
detail searches, 122
Detail view (Leaks instrument), 317-318
DEVELOPER_APPLICATIONS_DIR environment variable, 477
DEVELOPER_BIN_DIR environment variable, 477
DEVELOPER_DIR environment variable, 477
DEVELOPER_FRAMEWORKS_DIR environment variable, 477
DEVELOPER_LIBRARY_DIR environment variable, 477
DEVELOPER_TOOLS_DIR environment variable, 477
DEVELOPER_USR_DIR environment variable, 477
DEVELOPERSDK_DIR environment variable, 477
diagram view (ObjectAlloc instrument), 320
dictionary data type, 117-118
directories. *See also* bundles; packages
associating files with, troubleshooting, 113-114
for docsets, setting up, 230-231
framework directory structure, 210-211
for project products, 52
renaming in Subversion, 101
Directory I/O instrument, 451
directory structure for Spotlight plug-in project, 286
disabling breakpoints, 30, 390-391
discarding file revisions, 109
disclosure triangles
Option-clicking, 424
in project window, 15
Disk Monitor instrument, 455
displaying windows for custom views, 196
distccd daemon, 398
distributed builds, 397-399
docsets, 229. *See also* documentation
automating build process for, 233-235
as bundles, 230
copying files to, 231-233
directory setup, 230-231
document classes, 91
document file for sample application, 85-88
documentation. *See also* docsets
adding hierarchy to, 235-237
API documentation, creating, 238-241
Boolean text searches in, 471

creating
with Doxygen, 221
with HeaderDoc, 221-225
with user scripts, 225-229
HeaderDoc, HTML files generated by, 239
updating, 470-471
Documentation window, 502
enabling browsing, 235-237
downloading Xcode 3, 3
Doxygen, 221
drawing
custom views, 192-195
flushing graphics after, 199
to screen, modifying, 433
Dribin, Dave, 504
DTrace, 449, 456-457
DWARF debug information format, 376
dynamic libraries, 43-44. ***See also*** **frameworks**
defined, 490
location of, determining, 215
Dynamic Library target (BSD), 496
Dynamic Library target (Carbon), 496
Dynamic Library target (Cocoa), 497
Dynamic Library template, 490-491
dynamic loading of libraries, 43-44

E

editing
controller classes for data modeling, 256-258
data model files, 247
menu bars in projects, 301-302
object properties in Interface Builder, 123
scripts in Organizer window, 407-409
search scopes, 338
.subversion configuration directory, 96, 98
target settings, 53
user scripts, caution about, 228
windows, effect of resizing on, 71
Editing page (all-in-one layout), 350-351
editor panes, 331-332
Code Focus, 335-336
Code Sense in, 332-333
debugger strip, 336-337
jumping to symbol definitions, 333-334
multiple editor windows, 24
navigation bar, 334-335
opening, 22
preference modes, 337-338
emacs text editor, 469, 505
embedding
metadata in Core Data files, 291, 294-297
tools in Cocoa applications, 52-54
Empty Project template, 486
emptying outlets, 186
enabling
auto-properties in Subversion, 98
breakpoints, 390-391
browsing in Documentation window, 235-237
tracing, 433
encoding, types of, 312
end-of-file character (^D), 34
entities
creating, 259-266
in data modeling, 247
environment variables
list of, 477-478
viewing, 358
error bubbles, hiding, 23
error handling, NSAssert( ) macro, 192
error messages, parsing gcc error messages, 363
errors, build errors, 23-26
example build transcript, 359
Compile Sources build phase, 361-364
compiling data models, 364
compiling XIB files, 360
Copy Bundle Resources build phase, 361
copying structural files, 359-360
creating universal binaries, 365
Link Binary with Libraries build phase, 363-364

Run Script build phase, 360
Touch command, 365
executable files
custom executables, creating, 417
name of, 89
stripping. *See* stripping
Executable Info window, 388
EXECUTABLE_FOLDER_PATH build variable, 480
EXECUTABLE_NAME build variable, 479
EXECUTABLE_PATH build variable, 480
EXECUTABLE_PREFIX build variable, 479
EXECUTABLE_SUFFIX build variable, 479
Executables group, 310, 343-344
expression substitutions, 381
Extended Detail pane (trace document window), 444-445
extensions, 90
external build system projects, 413-416
Code Sense in, 416-417
debugging, 418
limitations of, 418-419
running, 417-418
External Build System template, 491-492
External Target target, 499

F

fast iteration in Objective-C 2.0, 173
faults, 450
favorites bar, 339
feature requests, Web site for, 503
File Activity instrument, 451, 459
file associations
setting for text editors, 468-469
troubleshooting, 113-114
File Attributes instrument, 451
file encoding, types of, 312
file formats for NIB files, 273-274
File Locks instrument, 450
file nodes, folder nodes versus, 237
file paths for macro-specification files, 134
file references, paths for, 469-470
file systems
instruments, list of, 450-451
resources and, 153
file-comparison window, 107-108
FileMerge, 111-112
filename extensions, 90
filenames
case sensitivity, 462
red color of, 113-114
files. *See also* data files; project files
assigning to framework targets, 205-210
copying to docsets, 231-233
deleting, 341, 406
list membership of, 354
merging, 111-112
moving, 406
renaming, 341, 406
saving before snapshots, 173
in targets, 354
Files List (Organizer window), actions in, 405-407
Files tab (condensed layout), 351
FileVault, Xcode performance and, 11
filtering
call trees, 444
man pages out of searches, 471
find/replace operations, 105-106
First Responder object, editing properties of, 123
Fix button (Debugger window), 32
fixing code in Debugger window, 35-37
flushing graphics, 199
folder nodes, file nodes versus, 237
folder references, creating, 466-467
Font key, 165
formal protocols in Objective-C, 58
format codes for x and print commands, 379
formats. *See* file formats; layout formats
formatters, debugging with, 379-381
forms
application windows, adding to, 68
labels, resizing, 307
resizing, 68
rows, adding, 68

Foundation Tool project, 490
frame rate for drawing screen, 434
Framework legacy target, 499
Framework target (Carbon), 496
Framework target (Cocoa), 497
Framework template, 491
frameworks, 203-204
adding from root file system, 270
debug frameworks, 374
defined, 491
directory structure of, 210-211
header files in, 205, 210
installation locations, 214-216
for private frameworks, 217-219
for public frameworks, 216-217
linking to projects, 211-213
system frameworks, 213-214
targets
adding, 204-210
assigning files to, 205-210
Info.plist for, 204-205
umbrella frameworks, 214
for unit testing, 167, 169
Frameworks directory (application bundles), 158
FRAMEWORKS_FOLDER_PATH build variable, 480
Full Screen toggle (trace document window), 445
functions, tail-recursive, 377

G

garbage collection
instruments, list of, 451
in Objective-C 2.0, 172
Garbage Collection instrument, 451
gatherheaderdoc utility, 222, 224-225
GC Total instrument, 451
gcc compiler suites, versions of, 5
gcc error messages, parsing, 363
GCC_ENABLE_OBJC_GC build variable, 481
GCC_PREPROCESSOR_DEFINITIONS build variable, 481
GCC_VERSION build variable, 481
GDBlog, 377
Generic C++ Plugin project, 490
Generic Kernal Extension legacy target, 500
Generic Kernel Extension project, 493
Generic Kernel Extension target, 498
GetMetadataForFile.c file in Spotlight plug-in project, 290-293
getter methods. *See* accessors
Getting Started tab (Welcome to Xcode window), 12
global searches in Organizer window, 410-411
global variables in Debugger window, 375
global-ignores setting (.subversion configuration file), 97
Globals Browser, 375
Go button (Debugger window), 31
Go command, 31
goals in makefiles, 353
graphics
flushing, 199
instruments, list of, 451
GraphWindow.xib file, localization, 308
GROUP environment variable, 477
Grouped/Ungrouped button, 337
grouping breakpoints, 376
groups, creating, 149
Groups & Files list, 339
adding columns to, 103
configuring, 345
Executables group, 343-344
groups, creating, 149
list membership of files, 354
organizing, 58
Project group, 339-341
Project Symbols smart group, 345
projects, placement in, 310
smart groups, 344-345
Targets group, 341-343

H

.h file suffix, 55
hardware acceleration, 434
header files
creating documentation from, 221-225
in frameworks, 205, 210

paths for, 462
precompiled headers, 361, 395-396, 464-465
HEADER_SEARCH_PATHS build variable, 482
HeaderDoc, 221-225
HTML files generated by, 239
user scripts and, 225-229
headerdoc2html utility, 222
Height key, 164
Hello, World project, 12-17
help. *See* resources
HFS API, 91
HFS type and creator, 91
HFS+ file system, case sensitivity of filenames, 462
hiding error bubbles, 23
hierarchical view of NIB files, 67-68
hierarchy
adding to documentation, 235-237
of build settings, 366-368
home directory, setting up for Subversion, 96-98
HOME environment variable, 477
hotkeys
removing, 448
setting, 448
HTML files, generated by HeaderDoc, 239
human interface, responding to, 51
human-interface events, replaying, 447
human-interface logging in Instruments application, 321-323

I

icon files, 91
identifiers, 89
Identity Inspector, 76
illegal operations in linear regression project, 29-30
Image Unit Plug-In for Objective-C project, 494
implementers, linking to when writing property lists, 125-126
importing
metadata, 297-299
projects to repository, 100-101
in-editor debuggers, Debugger window versus, 392
incrementing revision numbers, 104
index templates, 400
indexing
with Code Sense, 332-333
projects, 399-401
Info inspector, 470
Info windows for multiple items, 470
Info.plist file
in application bundles, 156
contents of, 158-165
copying in example build transcript, 359-360
for framework targets, 204-205
setting parameters for, 465
in Spotlight plug-in project, 288
InfoPlist.strings file, localization, 308-309
INFOPLIST_FILE build variable, 479
INFOPLIST_PREPROCESS build variable, 482
informal protocols, 58, 187
initialize class method, 191
initializers, redesigning data model, 249-250
injected tests. *See* dependent tests
input streams, terminating, 34
input/output instruments, list of, 451
Inspection Range tool in ObjectAlloc instrument, 319
Inspector palette (Interface Builder), 66
inspectors
dismissing, 447
Info inspector, 470
INSTALL_DIR build variable, 483
INSTALL_GROUP build variable, 483
INSTALL_MODE_FLAG build variable, 483
INSTALL_OWNER build variable, 483
INSTALL_PATH build variable, 483
INSTALL_ROOT build variable, 483

installation locations for frameworks, 214-216
private frameworks, 217-219
public frameworks, 216-217
Installer Plugin project, 494
installing
projects in Organizer window, 411-412
Xcode 3, 3-7
instantiation of top-level objects, retain count at, 182
instruments, 437. *See also* Instruments application
configuring, 446-447
custom instruments, 456-457
list of, 449-455
recording in, 447-448
saving trace documents, 449
Instruments application, 437. *See also* instruments
Library window, 445-446
MallocDebug and Shark compared, 437-438
memory leaks, checking for, 315-323
security, 316, 439
starting, 438-439
templates, 458-459
trace document windows. *See* trace documents, windows for
Instruments document, 321
Interface Builder, 63-66. *See also* NIB files
adding data sources and data points to created entities, 264-265
broken links and, 461-462
building interfaces for created entities, 260, 262-264
controller classes, editing for data modeling, 256-258
custom views, creating. *See* custom views
editing object properties in, 123
editing windows, effect of resizing on, 71
hierarchical view of NIB files, 67-68
Inspector palette, 66
layout functions of, 66-68
Library palette, 65-68
menu bars in projects, editing, 301-302
outlets, identifying, 182
parsing and, 461
resizing views, 69-72
splitting views, 72
version control and, 73
view classes, adding, 183
viewing data sources for created entities, 265-266
views, moving, 467
Interface Builder 3.x Plugin project, 494
internationalization. *See* localization
intrinsic libraries, linking, 463
inverse relationships, creating, 260
IOKit Driver legacy target, 500
IOKit Driver project, 493
IOKit Driver target, 498
iteration in Objective-C 2.0, 173

J–K

Java, Info.plist keys for, 163-164
Java Applet project, 492
Java Application project, 492
Java JNI Application project, 492
Java legacy target, 500
Java Signed Applet project, 492
Java target template, 498
Java template, 492-493
Java Tool project, 493
Java Web Start Application project, 493
JavaThread instrument, 455
jumping to symbol definitions, 333-334

Kernel Extension target template, 498
Kernel Extension template, 493
key-value coding (KVC) protocol, 50
key-value observing (KVO), 376
intercepting set accessors, 147
keyboard shortcuts. *See* hotkeys
keys (Info.plist)
list of, 159-164
localization of, 158
KVC (key-value coding) protocol, 50
KVO (key-value observing), 376
intercepting set accessors, 147

L

labels
changing with Interface Builder, 66
on controller objects, changing, 78
in forms, resizing, 307
languages, changing per application, 310. *See also* localization
latest version of files, base version versus, 108
launching. *See* starting
layout of sample application, 51, 63-73
editing windows, effect of resizing on, 71
with Interface Builder, 66-68
resizing views, 69-72
splitting views, 72
layout formats for projects, 348
all-in-one layout, 350-351
changing, 348
condensed layout, 351-352
default layout, 349-350
lazy loading, 418
lazy symbol loading, 384-385
leaks (memory usage), checking for
with Instruments application, 315-323
with MallocDebug application, 313-315
Leaks instrument, 316-317, 453-454, 459
call-tree view, 318-319
Detail view, 317-318
legacy targets, templates for, 499-500
libraries. *See also* dynamic libraries; static libraries
adding from root file system, 270
defined, 42
linking, 463
naming conventions, 142
prebinding, 45
Library legacy target, 499
Library palette (Interface Builder), 65-68
Library window (Instruments application), 445-446
LIBRARY_SEARCH_PATHS build variable, 482
LIBRARY_STYLE build variable, 479
line-ending styles in Debugger window, 377
linear regression. *See also* Cocoa applications
calculating, 20-22
command-line tool for, 22-23
debugging mode setup, 25-26
illegal operations in, 29-30
defined, 19
Link Binary with Libraries build phase, 43, 343
in example build transcript, 363-364
linking
frameworks to projects, 211-213
to implementers when writing property lists, 125-126
libraries, 463
outlets, 79-80
projects with repository, 104
source code, 42-46
weak linking, 271
links
broken links, Interface Builder and, 461-462
in NIB files, 75-77
load paths, references in bundles, 218
Loadable Bundle target (Carbon), 497
Loadable Bundle target (Cocoa), 497
loading symbols, 384-385
LOCAL_ADMIN_APPS_DIR environment variable, 477
LOCAL_APPS_DIR environment variable, 477
LOCAL_DEVELOPER_DIR environment variable, 477
LOCAL_LIBRARY_DIR environment variable, 477
Localizable.strings file, localization, 311-313
localization, 304
Credits.rtf file, 304-305
GraphWindow.xib file, 308
of Info.plist keys, 158
InfoPlist.strings file, 308-309
Localizable.strings file, 311-313
MainMenu.nib file, 305
MyDocument.nib file, 305-307

testing, 310-311
version control and, 307-308
locations
of dynamic libraries, determining, 215
for framework installations, 214-219
locking NIB file views, 272
locks instruments, list of, 455
logging
debugging log, clearing, 374
GDB log, 377
human-interface logging in Instruments application, 321-323
loops, testing conditions in, 34
LSBackgroundOnly key, 161
LSEnvironment key, 161
LSExecutableArchitectures key, 161
LSGetAppDiedEvents key, 161
LSHasLocalizedDisplayName key, 161
LSMinimumSystemVersion key, 161
LSMinimumSystemVersionByArchitecture key, 162
LSMultipleInstancesProhibited key, 162
LSPrefersCarbon key, 162
LSPrefersClassic key, 162
LSRequiresCarbon key, 162
LSRequiresClassic key, 162
LSRequiresNativeExecution key, 162
LSUIElement key, 162
LSUIPresentationMode key, 162
LSVisibleInClassic key, 162

M

.m file suffix, 55
machine instructions. ***See*** **assembly code**
The Mac Xcode 3 Book **(Cohen and Cohen), 501**
MacOS directory (application bundles), 157
macosx-dev mailing list, 503
macro-specification files, location of, 134
macros
in SenTestingKit framework, 168
text macros, 133-139
mailing lists for additional information, 502-503
main menu bar in NIB files, 90
MainHTML key, 165
MainMenu.nib file, localization, 305
makefiles, 353-354, 403
projects organized around, 404
external build system projects, 413-419
Organizer window, 405-413
preparation for, 404-405
MallocDebug application, 313-315, 433
Instruments application compared, 437-438
man pages, filtering out of searches, 471
managed-object model files, 245
master track instruments, 452
MAX_OS_X_DEPLOYMENT_TARGET build variable, 482
MAX_OS_X_VERSION_ACTUAL environment variable, 478
MAX_OS_X_VERSION_MAJOR environment variable, 478
MAX_OS_X_VERSION_MINOR environment variable, 478
mdls command-line tool, 281
measurement units, converting, 200
memory instruments, list of, 452-454
memory leaks, checking for
with Instruments application, 315-323
with MallocDebug application, 313-315
Memory Monitor instrument, 455
menu bars
localization, 305
in projects, editing, 301-302
merging files, 111-112
message invocation in Objective-C, 57-58
metadata. ***See also*** **Spotlight**
custom metadata keys, declaring, 288-290
embedding in Core Data files, 291, 294-297
importing, 297-299
UTIs, creating, 284-286
viewing, 281-284
methods
adding to classes, 118-121
invocation in Objective-C, 58

removing from projects to libraries, 146-147
responding to human interface, 51
unavailable methods, handling in cross-development, 268-270

MIME types, 90
Mini Debugger window, 389-391
Mini Instruments window, 448
model objects (MVC design pattern), 48
for sample application, 48-50
implementing, 54-62
Model phase (MVC design pattern), writing property lists, 118-121
Model-View-Controller (MVC) design pattern, 48
Controller phase, creating custom views, 181-183
for property lists, 118
adding categories to classes, 118-121
linking to implementers, 125-126
saving documents as property lists, 121-124
sample application
controller object for, 51, 75-92
implementing model classes, 54-62
model object for, 48-50
view object for, 51, 63-73
View phase, creating custom views, 183-187
models, class modeler tool, 346-348
modern bundles, 156
modification date, updating in example build transcript, 365
moving
files in Organizer window, 406
methods to libraries, 146-147
views in Interface Builder, 467
multiple architectures, testing for, 179-180
multiple editor windows, 24
multiple projects per repository, 98
multiple SDKs, cross-development with, 277-279
multiple-item Info windows, 470
MVC design pattern. *See* Model-View-Controller (MVC) design pattern
MyDocument class, redesigning data model, 255-256
MyDocument.nib file, localization, 305-307

N

naming conventions
for categories, 122
for libraries, 142
NaN (not a number), 29
NATIVE_ARCH environment variable, 478
NATIVE_ARCH32_BIT environment variable, 478
NATIVE_ARCH64_BIT environment variable, 478
navigation bar in editor panes, 334-335
nested scopes, visualization of, 335-336
Network Activity Monitor instrument, 455
New Class Model Assistant, 348
New Core Data Interface Assistant, 262
New Data Model File Assistant, 246
new features of Xcode 3, 1-2
New File Assistant, 142, 464
New Project Assistant, 12-13
New Standard Tool Assistant, 14
New Target Assistant, 141-142
New User Assistant window, 11-12
newsgroups for additional information, 503
NIB files
associating with projects, 183
compatibility, checking, 271-273
connecting buttons to, 84-85
controllers, adding to, 77-79
creating in New File Assistant, 464
custom classes, adding to, 76
defined, 63
formats for, 273-274
hierarchical view of, 67-68
links in, 75-77
localization, 305-307
main menu bar in, 90
opening in Interface Builder, 64

NIB loader, filling outlets, 197
nm tool, 149-150
nodes, folder nodes versus file nodes, 237
NSAppleScriptEnabled key, 163
NSArrayController class, 78
  arranged objects, 84
  bindings, 82
  deleting extra, 262
NSAssert( ) macro, 192
NSBundle class, 62
NSCoder Night Web site, 504
NSCoding protocol, 57, 185
NSController class, 77-78, 81
NSDocument class, 51
NSEntityDescription, 245
NSForm class, 68
NSHumanReadableCopyright key, 161
NSJavaNeeded key, 163
NSJavaPath key, 164
NSJavaRoot key, 164
NSMainNibFile key, 163
NSManagedObject, 245
NSManagedObjectContext, 245
NSManagedObjectModel, 245
NSMatrix class, 68, 464
NSMutableArray class, 50
NSNumberFormatter, 257-258
NSObjectController class, 78-82
NSPersistentDocument, 255
NSPrefPaneIconFile key, 164
NSPrefPaneIconLabel key, 164
NSPrincipalClass key, 160
NSServices key, 163
NSTask object, 50
NSZombieEnabled switch, 388-389
numeric values in user interface with Core Data, 257-258

O

Object Allocations template (Instruments application), 459
Object File target (BSD), 496
Object File target (Carbon), 497
Object File target (Cocoa), 497
object files, defined, 42
object properties, editing in Interface Builder, 123
object storage formats in Core Data, 256
OBJECT_FILE_DIR build variable, 480
OBJECT_FILE_DIR_normal build variable, 480
ObjectAlloc instrument, 316, 319-320, 433, 452-453
Objective-C
  debugging and, 36
  formal protocols, 58
  informal protocols, 58
  message invocation, 57-58
  method invocation, 58
Objective-C 2.0, 58
  built-in accessors, 50
  converting Cocoa projects to, 172-173
***Objective-C Pocket Reference* (Duncan), 502**
objects, connecting to outlets, 184-187
OBJROOT build variable, 479
online resources. *See also* resources
  mailing lists, 502-503
  Usenet newsgroups, 503
  Web sites, 503-504
opaque pointers in linear regression example library, 143
OpeGL Driver instrument, 451
opening
  console windows, 15
  editor window, 22
  Mini Debugger window, 389
  NIB files in Interface Builder, 64
  Target Info window, 53
OpenStep, 132
operating systems. *See* cross-development
optimization. *See also* performance tuning
  distributed builds, 397-399
  effect on order of operations, 41
  indexing, 399-401
  precompiled headers, 395-396
  predictive compilation, 396
  settings, 467-468
Option-clicking disclosure triangles, 424

organization name, setting default, 56
Organizer window, 405
benefits of, 419
configuring and building in, 409-411
Files List actions, 405-407
installing in, 411-412
running in, 412-413
script editing, 407-409
snapshots, creating, 409
toolbar for, 407-409
organizing Groups & Files list, 58, 149
orphan breakpoints, removing, 376
OS type codes, 91
Other tab (condensed layout), 352
OTHER_CFLAGS build variable, 481
OTHER_CFLAGS_normal build variable, 481
otool command, 150-151
outlets
actions and, 84-85
connecting to objects, 184-187
defined, 77
emptying, 186
filling with NIB loader, 197
identifying in Interface Builder, 182
linking, 79-80

P

p (print) command, 378-379
Package legacy target, 500
Package target (Java), 498
packages. *See also* bundles
explained, 153-154
RTFD package, 154-155
structured directory trees as, 91
viewing contents of, 154-155
PACKAGE_TYPE build variable, 478
packaging Spotlight plug-in project, 293-294
parsing
gcc error messages, 363
Interface Builder and, 461
paths
for file references, 469-470
for header files, 462
Pause button (Debugger window), 32
.pbxuser file, 463
Perforce, 94-95
performance tuning. *See also* optimization
with BigTop, 432
with CHUDRemover, 436
FileVault effect on, 11
with MallocDebug, 433
with ObjectAlloc, 433
optimization settings, 467-468
with Quartz Debug, 433-435
with Reggie SE, 432
with Sampler, 433
with Saturn, 432
with Shark. *See* Shark
with Spin Control, 435
with SpindownHD, 432
with Thread Viewer, 435-436
viewing background tasks, 332
PER_ARCH_CFLAGS build variable, 482
phases. *See* build phases
pixels, points versus, 434
PkgInfo file, copying in example build transcript, 359-360
playback head (Instruments application), 442
.plist files. *See* Info.plist files; property lists; XML property lists
plug-ins, Info.plist keys for, 163
Plugin key, 165
plutil tool, 129
po (print-object) command, 378
pointers
opaque pointers in linear regression example library, 143
released-pointer aliasing, 386-387
points, pixels versus, 434
prebinding libraries, 45
precompiled headers, 361, 395-396, 464-465
predictive compilation, 396
preference modes for editor panes, 337-338
preference panes, Info.plist keys for, 164
PreferencePane project, 494

preferences, setting, 466
prefix files, defined, 395
preparing makefile projects, 404-405
Preserve Bundle Contents check box (version control), 73
principal class for applications, 90
print (p) command, 378-379
print-object (po) command, 378
printing variable values, 377-379
private framework headers, 210
private frameworks, installation locations for, 217-219
Process instrument, 454
processor types, specifying in otool command, 151. *See also* cross-development
product directories for projects, 52
PRODUCT_NAME build variable, 478
products, version control and, 230
Products group, 310
profiling options in Shark, 431
project files for Spotlight plug-in project, 287
GetMetadataForFile.c, 290-293
Info.plist, 288
schema.strings, 290
schema.xml, 288-290
Project Find window, 105-106, 338-339
Project group, 339-341
project headers, 210
Project Info window, Target Info window versus, 215
Project Symbols smart group, 345
project templates, 485-486
Action, 486-487
Application, 487-489
Audio Units, 489
Bundle, 489
Command-Line Utility, 490
Dynamic Library, 490-491
Empty Project, 486
External Build System, 491-492
Framework, 491
Java, 492-493
Kernel Extension, 493
Standard Apple Plug-Ins, 493-495
Static Library, 495
project window, 14-15
PROJECT_DIR build variable, 479
PROJECT_FILE_PATH build variable, 479
PROJECT_NAME build variable, 478
ProjectBuilder IDE, 414
projects
building, cleaning before, 219
cleaning, 219
command-line utilities as, 14
committing files in version control, 110
comparing files in version control, 107
compiling, 395-399
configuring/building in Organizer window, 409-411
conflict resolution in version control, 110-113
defined, 12
dependent targets, 148-149
discarding file revisions, 109
files included in, 354
Hello, World project, 12-17
indexing, 399-401
installing in Organizer window, 411-412
layout formats, 348-352
libraries. *See* libraries
linear regression command-line tool, 22-26, 29-30
linking frameworks to, 211-213
menu bars, editing, 301-302
merging files, 111-112
multiple projects per repository, 98
NIB files, associating, 183
organized around makefiles, 404-419
placement in Groups & Files list, 310
product directories, 52
removing methods to libraries, 146-147
repository, adding to, 99-104
revising in version control, 105-113
rolling back revisions, 114-115
root directory for, 173

running in Organizer window, 412-413
saving before building, 23
selecting SDKs for, 267
Spotlight plug-in project. *See* Spotlight plug-in project
starting new, 52
tagging revision files, 115-116
targets, adding, 53, 141-142
working copies, checking out, 103-104
Xcode version compatibility, 279

properties. *See also* property lists
for Cocoa applications, 88-91
in Objective-C 2.0, 172

property accessors. *See* accessors

Property List Editor, 128-132

property lists, 117. *See also* Info.plist file; XML property lists
ASCII property lists, 132-133
binary property lists, 133
configuring colors from, 189, 191-192
data types in, 117-118
for user scripts, 229
viewing contents of, 127-132
writing, 118-126

protocols, informal, 187

public framework headers, 210

public frameworks, installation locations for, 216-217

public interface for linear regression example library, 143

Q–R

Quartz Composer Application project, 489

Quartz Composer Core Data Application project, 489

Quartz Composer Plug-In project, 494

Quartz Composer Plug-In with Internal Settings and User Interface project, 494

Quartz Debug, 433-435

Quartz Extreme, 434

Quick Look Plug-In project, 494

Quick Start keys. *See* hotkeys

quitting Xcode 3, 15

Reads/Writes instrument, 451

rebuilding
Code Sense indexes, 332
indexes, 400-401

recording in instruments, 447-448

red filenames, explanation for, 113-114

redesigning data model files, 247-256
accessors, 250-251
DataPoint class, 248-249
initializers, 249-250
Mydocument class, 255-256
Regression class, 251-255

refactoring, 171-174

references
file references, paths for, 469-470
folder references, creating, 466-467

Reggie SE, 432

registering repositories, 98-99

regression lines, defined, 19

Regression model class
redesigning data model, 251-255
source file, creating, 58-62

relationships, creating, 260

Release build configuration, 323-325, 368
Debug build configuration versus, 216, 275, 318

release notes, 502

released-pointer aliasing, 386-387

releasing top-level objects, 182

removing
files, 341, 406
hotkeys, 448
methods from projects to libraries, 146-147
NSArrayController, 262
orphan breakpoints, 376

renaming
directories in Subversion, 101
files, 341, 406

Rentzsch, Johnathan, 504

replaying
human-interface events, 447
human-interface traces, 322

Repositories window, 100
repository
adding projects to, 99-103
associating projects with, 104
checking out working copies of files, 103-104
committing changed files, 110
comparing files, 107
conflict resolution, 110-113
discarding file revisions, 109
registering, 98-99
revising files, 105-113
rolling back revisions, 114-115
setting up, 95-96
subdirectories in, 98
tagging revision files, 115-116
updates, 109
viewing contents, 103
Research Assistant window, 179, 240-241, 355
resizing
entity interfaces, 263
form labels, 307
forms, 68
views, 69-72
resolution, points versus pixels, 434
Resource File target (Carbon), 497
resource files, 153
resource fork, 153
Resource Manager, 153
resources
books, 501-502
Documentation window, 502
explained, 153-154
mailing lists, 502-503
text editors, 505
Usenet newsgroups, 503
user groups, 504
Web sites, 503-504
Resources directory (application bundles), 156
responder chains, 123
Restart button (Debugger window), 32
retain count of top-level objects, 182
revising project files
discarding file revisions, 109
rolling back revisions, 114-115
tagging revision files, 115-116
in version control, 105-113
revision numbers, incrementing, 104
roles for applications, 91
rolling back file revisions, 114-115
root directory for projects, 173
root file system, adding libraries/frameworks from, 270
rows, adding to forms, 68
RTFD package, 154-155
Ruby Extension project, 491
Ruby target template, 498
rules, creating custom build rules, 356-357
Run command, 31
Run Script build phase, 233-235, 342, 357-358, 360
run scripts, creating in Organizer window, 411-412
running. *See also* starting
applications, 31
Core Data data files, 258-259
external build system projects, 417-418
Hello, World project in Terminal application, 16
linear regression example library, 152
projects in Organizer window, 412-413
unit tests, 175-176

S

sample application. *See* Cocoa applications; linear regression
Sampler instrument, 433, 454
Saturn, 432
saving
documents as property lists, 121-124
files before snapshots, 173
projects before building, 23
trace documents, 449
schema.strings file in Spotlight plug-in project, 290

schema.xml file in Spotlight plug-in project, 288-290
SCM (software configuration management). *See* **version control**
SCM Results window, 350
scopes, editing search scopes, 338
screen, modifying drawing to, 433
Screen Saver project, 494
Script menu, generating property accessors, 50
scripts
- automating docset build process, 233-235
- default scripts in Organizer window, 413
- editing in Organizer window, 407-409
- Run Script build phase, 233-235, 342, 357-358, 360
- run scripts, creating in Organizer window, 411-412
- user scripts, 225-226

SDK targets, setting, 270
SDKROOT build variable, 479
SDKs for cross-development, 267-270, 277-279
search paths, list of, 482
search scopes, editing, 338
searches
- Boolean text searches in documentation, 471
- detail searches, 122
- filtering man pages out of, 471
- global searches in Organizer window, 410-411
- with Project Find window, 338-339

security, Instruments application, 316, 439
selecting
- project SDKs, 267
- target SDKs, 268

SenTestingKit framework, 167, 169
set accessors, Key-Value Observing protocol and, 147. *See also* **accessor methods**
setting names for build variables, 475
setting titles for build variables, 476
settings hierarchy for build system, 366-368
Shared Libraries window, 385
Shared Memory instrument, 452
Shared Workgroup builds, 397-399
sharing precompiled header files, 464-465
Shark, 320, 421-422
- callstack data mining, 425-427
- Instruments application compared, 437-438
- optimizing Linear Regression example, 428-431
- starting, 422
- viewing analysis, 423-425

***Shark User Guide*, 502**
Shell Script Automator Action project, 487
Shell Script Target target, 499
Shell Script targets, creating, 233-235
Shell Tool target (BSD), 496
Shell Tool target (Carbon), 497
Shell Tool target (Cocoa), 497
Shipley, Wil, 504
shortcut keys. *See* **hotkeys**
single build directory, setting preferences for, 465
singularity, avoiding, 302-304
Size inspector, 69
sizing. *See* **resizing**
smart groups, 344-345
snapshots, 173-174
- creating in Organizer window, 409
- saving files before, 173
- version control versus, 174

software configuration management (SCM). *See* **version control**
source code
- compiling, 39-41
- defined, 39
- linking, 42-46
- for sample application controller, 85-88
- viewing for HeaderDoc-generated HTML files, 239

source files
for DataPoint model class, creating, 55-58
for Regression model class, creating, 58-62
source locations, list of, 479
source trees, 483
source-code management (SCM). *See* version control
Special Targets target template, 499
speed. *See* optimization
Spin Control, 435
Spin Monitor instrument, 454
SpindownHD, 432
splitting views, 72
Spotlight importers, 284, 286
Spotlight Plug-In project, 494. *See also* metadata
directory structure, 286
packaging, 293-294
project files in, 287-293
template for, 286
testing, 297-299
troubleshooting, 294
verifying, 294
version control, 286-287
SQL format (Core Data), 256
SRCROOT build variable, 479
STABS debug information format, 376
Standard Apple Plug-Ins template, 493-495
standard error stream, writing to Console, 377-379
Standard Tool project, 490
starting. *See also* running
Instruments application, 438-439
new projects, 52
Shark, 422
Xcode 3, 11
static class models, 346
static libraries, 43
adding as targets, 141-142
defined, 495
designing, 143-146
limitations of, 203
moving methods to, 146-147
running, 152
stripping dead code, 327-328
verifying contents of, 149-151
Static Library target (BSD), 496
Static Library target (Carbon), 497
Static Library target (Cocoa), 497
Static Library template, 495
statistical profilers, 421
Step Into button (Debugger window), 32
***Step into Xcode: Mac OS X Development* (Anderson), 501**
Step Out button (Debugger window), 32
Step Over button (Debugger window), 32
stepping through code, 33-35
Stevenson, Scott, 504
Stop button (Debugger window), 31
storage types (Core Data), 91
stripping, 325
dead code, 327-328
symbol tables, 326-327
structured directory trees, as packages, 91
struts, 69
Style menu (instrument configuration), 446
subdirectories in repository, 98
SubEthaEdit editor, 505
subgroups, 340
substitution markers, 227-228
Subversion, 94
branching in, 243-245
changed recorded by, 149
directories, renaming, 101
home directory, setting up, 96-98
repository. *See* repository
revision numbers, incrementing, 104
.subversion configuration directory
creating, 96
editing, 96-98
symbol definitions, jumping to, 333-334
symbol tables, 149-150, 325-327
symbolic breakpoints, setting, 373
symbols
defined, 41
documentation for, 238-241

lazy loading, 384-385, 418
Project Symbols smart group, 345
refactoring, 174
SYMROOT build variable, 479
Sync Schema project, 494
system frameworks, 213-214
system instruments, list of, 454-455
system tables, debugging information versus, 326

T

tables, adding to application windows, 67
tagging
HeaderDoc support for, 222
revision files, 115-116
tags directory, 99
tail-recursive functions, 377
tarballs, 403-405
Target Info window
Build tab, 177
opening, 53
Project Info window versus, 215
target templates, 495-496
BSD, 496
Carbon, 496-497
Cocoa, 497-498
Java, 498
Kernel Extension, 498
for legacy targets, 499-500
Ruby, 498
Special Targets, 499
target-dependency-action group, 354
TARGET_BUILD_DIR build variable, 480
TARGET_NAME build variable, 478
targets
active targets, 141, 148
build configurations versus, 92
build phases for, 41, 342
build targets, list of, 478-479
building, 15
cleaning before building, 268
defined, 12
dependent targets, 148-149
deployment targets, setting, 270
editing settings for, 53
files in, 354
framework targets, 204-210
product types in, 354
projects, adding to, 53, 141-142
SDK targets, setting, 270
selecting SDKs for, 268
Shell Script targets, creating, 233-235
unit test targets, creating, 167-171
Targets group, 310, 341-343
Targets tab (condensed layout), 352
technical support. *See* resources
templates. *See also* project templates; target templates
index templates, defined, 400
in Instruments application, 458-459
for Spotlight plug-in project, 286
Terminal application, running Hello, World project in, 16
terminating input streams, 34
TesseractOCR, 404
in external build system project, 413-418
Organizer window, 405
configuring and building in, 409-411
Files List actions, 405-407
installing in, 411-412
running in, 412-413
script editing, 407-409
snapshots, creating, 409
toolbar for, 407-409
unpacking from archive, 404-405
test classes, creating, 168
testing. *See also* unit testing
custom views, 196-198
localization, 310-311
loop conditions, 34
Spotlight plug-in project, 297-299
universal binaries, 276-277
text, viewing property list contents as, 127-129
text editors
file associations, setting, 468-469
list of, 505

text files, treating .xcodeproj package as, 97-98
text macros, 133-139
TextMate editor, 505
TextWrangler editor, 505
Thread Viewer, 435-436
threads instruments, list of, 455
three-way file merges, 111-112
Time Analysis window (Shark), 423-425
Time Profile (All Thread States) mode (Shark), 431
Time Profile (WTF) mode (Shark), 430
to-many relationships, creating, 260
Tokens.xml file, 238-241
toll-free bridging, defined, 389
Tool legacy target, 500
Tool target (Java), 498
toolbar buttons. *See* buttons
toolbars
  Organizer window, 407-409
  trace document windows, 439-441
tools, embedding in Cocoa applications, 52-54
top-down view (Shark analysis results), 425
top-level objects, retain count at instantiation, 182
Touch command in example build transcript, 365
trace documents, 438
  saving, 449
  window for, 439
    Action menu, 445
    Detail pane, 442-444
    Extended Detail pane, 444-445
    Full Screen toggle, 445
    toolbar, 439-441
    Track pane, 441-442
traces, marking with human-interface events, 321-323
tracing, enabling, 433
Track pane (trace document window), 441-442
tracking class models, 346
translation. *See* localization
troubleshooting. *See also* resources
  broken links, 461-462
  DTrace crashes, 457
  file associations, 113-114
  file encoding, 312
  full-screen mode (Instruments application), 445
  indexing, 399-401
  library links, 463
  parsing problems, 461
  Spotlight plug-in project, 294
  XML property lists, 129
trunk directory, 99
tuning. *See* optimization; performance tuning
type codes, 89
Type menu (instrument configuration), 447

U

UI Recorder template (Instruments application), 459
umbrella frameworks, 214
unarchiving custom views, 185
unavailable methods in cross-development, handling, 268-270
undoing. *See* rolling back
uniform type identifiers. *See* UTIs
uninstalling Xcode 3, 7-8
Unit Test Bundle target (Carbon), 497
Unit Test Bundle target (Cocoa), 498
Unit Test Target target (Ruby), 498
unit testing, 167
  dependent tests, 176-179
  multiple architectures, 179-180
  refactoring, 171-174
  running tests, 175-176
  targets, creating, 167-171
units of measurement, converting, 200
universal binaries, 274-276
  creating in example build transcript, 365
  Intel-porting issues in Linear example, 276
  testing, 276-277

UNLOCALIZED_RESOURCES_FOLDER_PATH build variable, 480
unpacking tarballs, 404-405
updating
documentation, 470-471
modification date in example build transcript, 365
in Subversion, 109
Usenet newsgroups for additional information, 503
USER environment variable, 477
user groups for additional information, 504
user interface instruments, list of, 455
user interface traces. *See* human-interface traces
User Interface track, 452. *See also* human-interface logging
user scripts
%%%{PBX}%%% substitution markers, 227-228
creating documentation with, 225-229
editing, caution about, 228
UTExportedTypeDeclarations key, 160
UTImportedTypeDeclarations key, 160
UTIs (uniform type identifiers), 90, 283-286

V

VALID_ARCHS build variable, 481
validation of data, 302-304
value binding, 82-83
variables. *See also* build variables
environment variables, viewing, 358
global variables in Debugger window, 375
printing values of, 377-379
viewing with data formatters, 379-381
verifying
library contents, 149-151
Spotlight plug-in project, 294
version compatibility in Xcode projects, 279
version control, 27, 93-94
branching, 243-245
CVS (Concurrent Versions System), 94
for docset directories, 230-231
home directory, setting up for Subversion, 96-98
Interface Builder and, 73
localization and, 307-308
Perforce, 94-95
products and, 230
for property list files, 120
repository. *See* repository
revision numbers, incrementing, 104
snapshots versus, 174
for Spotlight plug-in project, 286-287
Subversion, 94, 149
Xcode support for, 94-95
of XIB files, 274
***Version Control with Subversion* (Collins-Sussman, Fitzpatrick, Pilato), 501**
versioned bundles, 156
vertical layout in Debugger window, 373
vi editor, 505
view classes, adding, 183
view objects (MVC design pattern), 48, 51, 63-73
View phase (MVC design pattern)
creating custom views, 183-187
writing property lists, 121-124
viewing
background tasks, 332
build settings, list of, 466
build variables, 475-476
data sources for created entities, 265-266
environment variables, 358
Info.plist file contents, 158-165
metadata, 281-284
package contents, 154-155
property list contents, 127-132
repository contents, 103
Shark analysis results, 423-425
source code for HeaderDoc-generated HTML files, 239
variables with data formaters, 379-381

views. ***See also*** **custom views**
layout of, 66-68
moving in Interface Builder, 467
resizing, 69-72
splitting, 72
visualization of nested scopes, 335-336

W

WARNING_CFLAGS build variable, 482
watchpoints, setting, 375
weak linking, 271
Web sites for additional information, 503-504
WebKit Plug-In project, 495
WebOjbects.mpkg, 4
Welcome to Xcode window, 12-13
widgets, Info.plist keys for, 164-165
Width key, 164
Windowed Time Facility (WTF), 430
windows. ***See also*** **editor panes; Groups & Files list; trace documents, windows for**
adding interface elements to, 66-68
Build Results window, 349, 359
changing default layout, 464
Class Browser window, 345-346
Console window, 349-350
displaying for custom views, 196
editing, effect of resizing on, 71
favorites bar, 339
Info windows for multiple items, 470
Library window (Instruments application), 445-446
Mini Instruments window, 448
Project Find window, 338-339
Research Assistant window, 355
SCM Results window, 350
working copies, checking out, 103-104
WRAPPER_NAME build variable, 479
writing property lists, 118
adding categories to classes, 118-121
linking to implementers, 125-126
saving documents as property lists, 121-124
WTF (Windowed Time Facility), 430

X–Z

x command, 378-379
Xcode, version compatibility, 279
Xcode 2.5, 8
Xcode 3
downloading, 3
installing, 3-7
launching, 11
new features, 1-2
obtaining, 3
quitting, 15
uninstalling, 7-8
Xcode News tab (Welcome to Xcode window), 12
Xcode User Guide**, 502**
xcode-users mailing list, 503
xcodebuild tool, 365-366
.xcodeproj package, treating as text, 97-98
XcodeTools.mpkg, 3
xed utility, 224
XIB files, 273-274
compiling in example build transcript, 360
creating in New File Assistant, 464
XML format (Core Data), 256
XML property lists, 128
Code Focus and, 336
creating text macros for, 133-139
troubleshooting, 129

ZERO_LINK build variable, 482
ZeroLink, 45-46
zombies, debugging with, 385-389
Zoom slider (instrument configuration), 447