Programming Mac OS X: A GUIDE FOR UNIX DEVELOPERS

KEVIN O'MALLEY

MANNING

Programming Mac OS X

Programming Mac OS X

A GUIDE FOR UNIX DEVELOPERS

KEVIN O'MALLEY

MANNING
Greenwich
(74° w. long.)

For electronic information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue Fax: (203) 661-9018
Greenwich, CT 06830 email: orders@manning.com

©2003 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Tiffany Taylor
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-930110-85-5

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 05 04 03 02

brief contents

PART 1 OVERVIEW 1

1 ■ Welcome to Mac OS X 3

2 ■ Navigating and using Mac OS X 27

PART 2 TOOLS 55

3 ■ Project Builder and Interface Builder 57

4 ■ Development tools 109

PART 3 PROGRAMMING 169

5 ■ Objective-C and the Cocoa development frameworks 171

6 ■ Cocoa programming 203

7 ■ AppleScript programming 245

8 ■ Mac OS X and beyond 279

contents

foreword xiii
preface xv
acknowledgments xviii
about this book xix
about the author xxiii
about the cover illustration xxiv

PART 1 OVERVIEW 1

1 ***Welcome to Mac OS X*** *3*

1.1 Introduction *4*

Origins of Mac OS X *5*

1.2 The Macintosh user interface *6*

1.3 The Mac OS X user interface *8*

The desktop *8* ▪ *Menus* *8* ▪ *The Dock* *10*
Window layering *11* ▪ *Dialog boxes* *11* ▪ *Drawers* *12*
Keyboard navigation *12* ▪ *Other interface features* *13*

1.4 The Mac OS X architecture *13*

Architecture layers *15* ▪ *The kernel environment* *16*
Core Services layer *20* ▪ *Application Services layer* *21*
Application Environment layer *22* ▪ *Aqua* *26*

1.5 Summary *26*

2 *Navigating and using Mac OS X 27*

2.1 Introduction *28*
2.2 Shells *29*
Terminal features 31
2.3 Help system *32*
Help Viewer 33
2.4 User accounts and privileges *33*
Creating user accounts 34
2.5 Booting and default services *36*
2.6 Programs and Mac OS X bundles *37*
2.7 Security issues *39*
2.8 File system *39*
Finder 41 ▪ *Case sensitivity and pathname delimiters 43*
2.9 Single-user mode *44*
2.10 System log files *45*
2.11 Processes management *45*
2.12 Common commands and tools *46*
2.13 Scripting languages *48*
AppleScript 48
2.14 Development tools *50*
2.15 X Window under Mac OS X *51*
Installing the X server 52
2.16 UNIX to Mac OS X software projects *53*
2.17 Summary *54*

PART 2 TOOLS 55

3 *Project Builder and Interface Builder 57*

3.1 Introduction *58*
Macintosh Programmer's Workbench 59
THINK Pascal and THINK C 59 ▪ *CodeWarrior 60*
Project Builder and Interface Builder 60
3.2 Creating an application with Project Builder *62*

3.3 Project Builder in depth *67*

Targets and build styles 67 ▪ Project Builder's UNIX tools 68
Project Builder's interface 69 ▪ Project Builder scenarios 78

3.4 Creating an application with Interface Builder *100*

Interface Builder scenarios 101

3.5 Summary *108*

4 ***Development tools 109***

4.1 Introduction *110*

4.2 UNIX development tools under Mac OS X *112*

Editors 112 ▪ Mac OS X editing tools 113
Version control 117 ▪ Static code analysis tools 121

4.3 Compilers and build tools *122*

4.4 Mac OS X Aqua-based development tools *122*

UNIX-based editors 122 ▪ Mac OS X-based editors 127

4.5 Apple's GUI-based development tools *127*

Apple Help Indexing Tool 128 ▪ AppleScript Studio 128
FileMerge 129 ▪ Icon Composer 132
Interface Builder 132 ▪ JavaBrowser 133
MRJAppBuilder 134 ▪ MallocDebug 135 ▪ ObjectAlloc 143
PEF Viewer 143 ▪ PackageMaker 144 ▪ Pixie 144
Project Builder 144 ▪ PropertyListEditor 144
Quartz Debug 146 ▪ Sampler 147 ▪ Thread Viewer 150
icns Browser 155

4.6 Apple's command-line development tools *156*

ps (process status) and top (system usage statistics) 156
sc_usage: showing system call usage statistics 158
fs_usage: reporting system calls and page faults related to the filesystem in real-time 160 ▪ gprof: displaying execution profile data 161 ▪ leaks: searching a process's memory for unreferenced malloc buffers 163 ▪ heap: listing all the malloc-allocated buffers in the process's heap 165
malloc_history: showing malloc allocations that a process has performed 165 ▪ sample: profiling a process during a time interval 166

4.7 Summary *167*

PART 3 PROGRAMMING 169

5 *Objective-C and the Cocoa development frameworks 171*

5.1 Introduction *172*

5.2 Introduction to Objective-C *173*

Object-oriented terminology 174 ▪ Classes 175
Messages 177 ▪ Categories 178 ▪ Protocols 180
Other features 180 ▪ Why learn Objective-C? 181

5.3 Cocoa software infrastructure *182*

Foundation 182 ▪ Application Kit 187
Memory management 188 ▪ Design patterns 193
Cocoa event handling 197

5.4 Other Cocoa development languages *200*

C++ 201 ▪ Perl 201 ▪ Ruby 202

5.5 Summary *202*

6 *Cocoa programming 203*

6.1 Introduction *204*

6.2 The CocoaWGet example program *205*

6.3 Program requirements *207*

6.4 Program design *208*

6.5 Building the interface *209*

Opening the project 209 ▪ The interface components 210
Control alignment and spacing 212 ▪ Forms 215
Classes and instances 215

6.6 CocoaWGet: implementing code with Project Builder *220*

The model 221 ▪ The view 224 ▪ The controller 224

6.7 Program extensions *233*

Letting the user cancel downloads 234
The application icon 239 ▪ The help file 241

6.8 Summary *243*

7 *AppleScript programming 245*

7.1 Introduction *246*

7.2 Scripting languages *247*

7.3 AppleScript *248*

Creating and running a script 250 ▪ Types of AppleScripts 251
AppleScript extensions 252 ▪ The AppleScript language 254
Choosing a scripting language 264

7.4 Example applications of AppleScript *264*

iTunes and AppleScript 264 ▪ AppleScript Studio 269

7.5 Summary *278*

8 *Mac OS X and beyond 279*

8.1 Introduction *280*

8.2 Development tools *281*

Compilers 281 ▪ Project Builder 283 ▪ Changing compilers 283
Inline scripting 283 ▪ New target editor 286
Searching documentation 287

8.3 Terminal application *289*

Setting Terminal preferences 289
Splitting the Terminal window 292
Other Terminal additions 293

8.4 The PerlObjCBridge *293*

PerlObjCBridge example 295

8.5 Summary *300*

A *Getting and installing development tools 301*

B *UNIX and Mac OS X command mappings 303*

B.1 Common Mac OS X operations *304*

B.2 UNIX file/directory commands mapped to Mac OS X commands *304*

List directory contents: ls 304
Copy/move files or folders: cp, mv 305
Remove files or folders: rm 305 ▪ Change directory: cd 305

Create a new directory: mkdir 305
Change file permission and group: chmod, chgrp 306
Compare files: diff 306 ▪ Get the word, line, or byte count: wc 306
Compress and decompress data: compress, uncompress, tar, gzip, gnuzip, unzip, zcat 306 ▪ Edit text files: emacs, vi 306
View files: head, tail 306 ▪ Find files: find 307

B.3 UNIX communication commands mapped to Mac OS X commands *307*

OpenSSH: ssh, scp 307 ▪ Talk to another user: talk, ytalk 307

B.4 UNIX process management commands mapped to Mac OS X commands *307*

Show system and process usage statistics: top, ps 307
Terminate a process: kill 307

C ***The precursor of Mac OS X: Mac OS 309***

C.1 A tour of the Mac OS interface *310*

C.2 Interacting with the system *312*

C.3 Mac OS system components *313*

System file and Finder 314 ▪ Process scheduling 314
Memory management 315
Extending the system through system extensions 317
Interapplication communication (IAC) 318
File system 319 ▪ Macintosh files 319 ▪ Graphics 320
Networking 321

D ***A brief history of UNIX 323***

D.1 The origin of UNIX *324*

High-level languages and punch cards 324
Batch processing 325 ▪ Time-sharing 326

D.2 The birth and development of UNIX *328*

D.3 GNU, Free Software Foundation, and open source *333*

D.4 UNIX software development philosophy *335*

resources 337

index 345

foreword

Apple's release of the Macintosh in 1984 heralded a computer revolution in ease of use for nontechnical people. Over time, computers and computer interfaces split into three main camps: Microsoft Windows, the Macintosh, and the various flavors of UNIX.

UNIX has been a traditional favorite of the research and scientific community for a variety of reasons. With the rise of Linux, it has become more popular than ever. Now Apple has brought the worlds of Macintosh user experience and UNIX together to form Mac OS X. With a full-featured UNIX system as the driving engine, the two worlds have merged.

All that remains is to create better software for this new blended environment. This has proven to be challenging. Many UNIX developers haven't written code for graphical user interfaces, while many Macintosh developers haven't written code based on UNIX environments. Bringing these two diverse types of developers to the same playing field can be difficult because they each need to learn from the other.

This book is a large step forward in facilitating that combined knowledge. While introducing UNIX developers to the tools available under Mac OS X at a favorite price point (i.e., free), it also shows Macintosh developers how to adapt to this new environment and make the most of the new tools now available to them.

This book is a clear roadmap for learning to write software under Mac OS X. As a longtime Macintosh developer (with a little UNIX experience), I can say

this with confidence. I got the chance to read this book just as I was making the transition from MacOS 9 to Mac OS X. It has helped my understanding of this new environment by refocusing my UNIX knowledge to this new target.

For the experienced UNIX developer, this book is your native guide to the Mac OS X landscape. It speaks both your language and the language of the natives, helping you quickly make the transition to Mac OS X development.

While the transition from UNIX to Mac OS X may seem daunting, this book is a gentle guide, highlighting the development issues found along the way and smoothing the sometimes serpentine path of coding we all travel.

SHANE LOOKER
Senior Software Engineer
Electronics for Imaging, Inc.

In the mid-1980s, Apple came up with some fundamental principles for how the Macintosh and its applications should look and feel: the Macintosh Human Interface Guidelines. The goal was to present users with a powerful, consistent system that was easy to use and that had an uncomplicated user interface. These design goals centered on the user being in charge of the computer and advocated techniques such as direct feedback for user actions, use of real-world metaphors for user interface elements, and a consistent user interface shared between and among applications. (Remember, these were the days when most personal computers ran MS-DOS and users interacted with the system using a command prompt and text-based interfaces.)

For example, imagine you were developing an application and working on its user interface. One method would be to design your application's interface from scratch according to your own preferences, or possibly base it on a similar program's interface and make appropriate modifications. Now imagine if developers built all applications this way. The result would be applications that look and behave very differently and implement common operations in dissimilar ways. The consequence for users would be an uneven user experience and constant relearning of tasks when moving to new applications.

Macintosh programmers did things differently. Instead of designing and laying out their applications' user interface any way they wished, they followed the guidelines Apple provided them; this process ensured that applications maintained the Macintosh look and feel. In addition, Apple's toolbox routines did much of the work of supporting that interface—for most developers, breaking the guidelines involved more work than following them. At first this programming approach was quite a shift, and it probably would not have succeeded if the guidelines had not been well thought out or did not make sense. Luckily, Apple employed some smart, experienced people who cared a great deal about how users interact with computers. The Macintosh Human Interface Guidelines became a cornerstone for user interface development on the Macintosh, and most applications were judged and evaluated based on these principles.

The consequences of these guidelines are applications that implement interface elements and standard operations in a consistent way, enabling users to easily translate their current knowledge to new programs. Over the years, the interface guidelines have grown as new technologies and interface components have been added to the Macintosh system. Today, the Aqua Human Interface Guidelines (http://developer.apple.com/techpubs/macosx/Essentials/AquaHIGuidelines) describe how to construct user interfaces for Mac OS X applications. To a degree,

the Aqua guidelines are another extension of the original interface guidelines, addressing new features of the Mac OS X user interface.

The most important lesson to take from this discussion is that Apple has put a lot of time and thought into how Macintosh applications should look and behave. The company has produced an excellent set of rules and recommendations for constructing contemporary user interfaces, and developers should read, understand, and follow them when developing Macintosh applications. Try to envision the programs you write for Mac OS X as being members of a complete, well-thought-out system where certain rules exists to promote the user experience. Your application should exist within this context, and not as a separate entity.

1.3 The Mac OS X user interface

The strength of the Macintosh has always been its user interface and ease of use. The new Mac OS X Aqua interface maintains the tradition of intelligent, easy-to-use Macintosh user interfaces, but sports a distinctive, liquid-like look, as well as many new and advanced interface components and features. Figure 1.2 shows an example of the Aqua user interface.

The Aqua interface continues to use real-world metaphors to represent computer resources. Navigating and using the system is simple because you are already familiar with many of these concepts. Overall, the Aqua user interface is simple and intuitive compared to UNIX desktops and window managers such as GNOME (http://www.gnome.org), KDE (http://www.kde.org), and fvwm (http://www.fvwm.org). As a result, you will require little upfront information to begin using the system.

1.3.1 The desktop

The Mac OS X desktop is analogous to a real office desk, which functions as your primary workspace and repository of information. A program called the *Finder* works with the system software to provide users with file management and process invocation functions, and presents and manages the desktop.

1.3.2 Menus

Under Aqua, an application displays its menu bar at the top of the screen. This is different from Windows or UNIX environments, where the menu bar appears at the top of each application window. The items in the menu bar are ordered as follows (from left to right): Apple menu, application menu, application-defined menus, window menu, help menu, and menu status bar items (see figure 1.3).

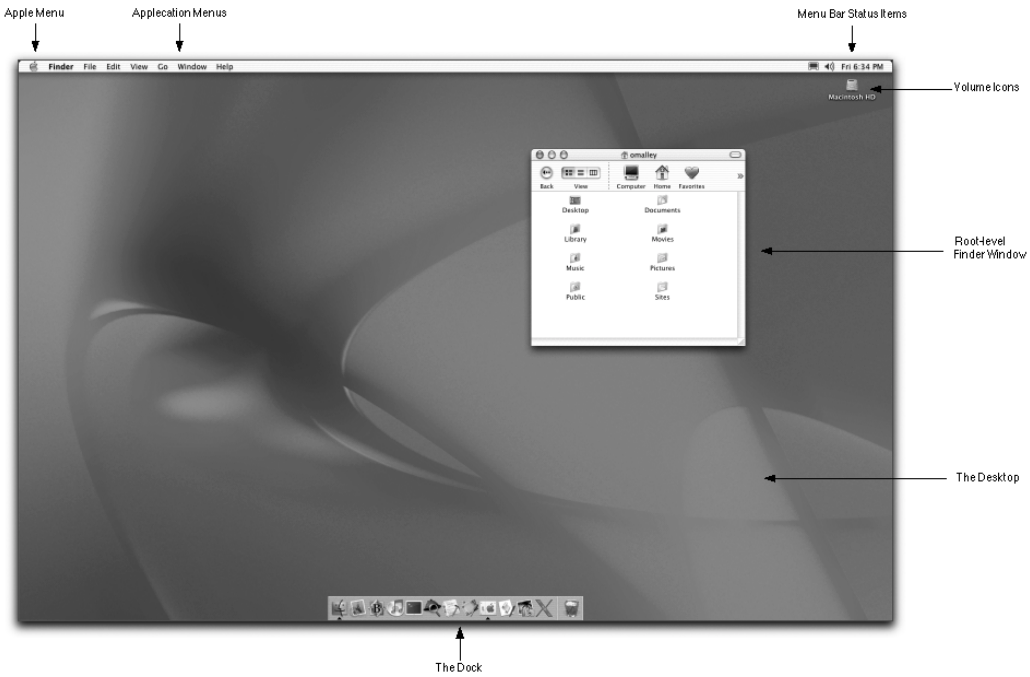

Figure 1.2 Aqua, the user interface for Mac OS X, builds on many features of the original Macintosh user interface. However, it has an entirely new look and feel, as well as many new features.

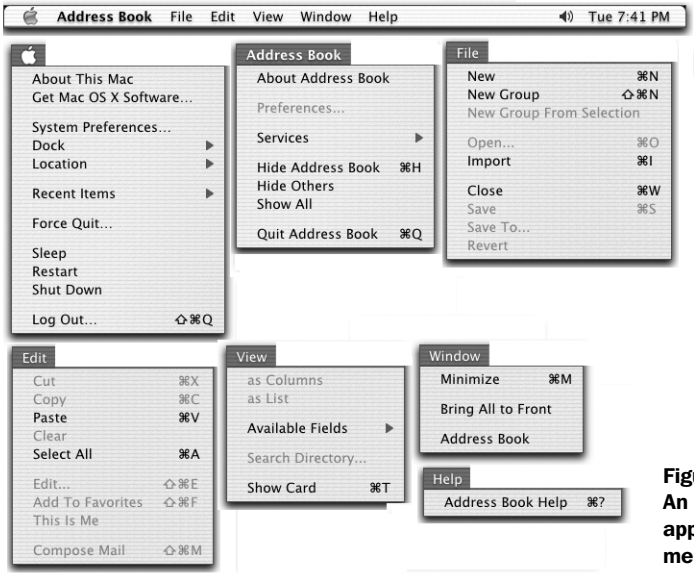

Figure 1.3
An example of a Mac OS X application's (Address Book) menu bar and menu items

First is the Apple menu, a system-wide menu whose contents do not change. Its commands permit users to perform tasks that operate on the system as a whole and are independent of any particular application. Commands support accessing system preferences, restarting and shutting down the computer, and logging off the current session.

Next is the Application menu, which holds items that apply to a specific application. Menu items include the application's preferences, services provided by other applications, and the Quit option. The menu name is bold, so it stands out from the other menus.

The next set of menus is application defined, but it typically includes the following menus, in this order: File and Edit, application-defined menus (possibly including View), Window and Help. They perform these functions:

- The File menu implements operations for document management such as opening, creating, and printing documents.
- The Edit menu contains commands for editing application documents and sharing application data over the clipboard.
- The View menu holds commands enabling users to change or alter the view of an application's current window.
- The Window menu lists currently open windows as well as window operations.
- The Help menu provides access to application help.
- Status items appear as the final, rightmost menu item and display information about system services, enabling quick access to system settings.

NOTE *Clipboard* is a Macintosh term for a common shared data holder used by the applications to temporarily hold data or to transfer data from one application to another. On the Macintosh, terms like *copy*, *cut*, and *paste* describe editing operations. For example, after you highlight an item in a document, you can perform a cut, which moves the selected item from the document to the clipboard; a copy, which copies the selected item to the clipboard; or a paste, which copies the item on the clipboard to the desired location.

1.3.3 The Dock

The *Dock*, located at the bottom of the screen in Figure 1.2, is a small toolbar that provides a standard, system-supplied location for you to organize commonly

accessed items such as applications, documents, and other information. It also aids in maneuvering between running applications.[1]

You add items by dragging their icons to the Dock; you remove items by dragging them off the Dock. Clicking an icon will bring it to the foreground, launching it first if it is not already running. A triangle next to an application icon indicates that the application is running. The Dock also holds the familiar Macintosh Trash icon, which collects files waiting to be deleted from the system. You can customize the Dock's appearance and behavior through the System Preference program, located in /Application.

1.3.4 Window layering

The original Mac OS imposed a window-layering scheme that placed all application windows conceptually on a single layer. This meant that if you were using one application and you clicked a window from another application, all of that application's windows came to the foreground. Mac OS X implements a different window-layering model: windows within an application are independent of one another, and can therefore be interleaved with windows from different applications.

Imagine you have two applications running, each with several visible windows. Under Mac OS X, only the window you click comes to the foreground, enabling windows from different applications to be interspersed. The result is more information simultaneously visible at a time and fewer visible transitions between applications. Perceptually, the new window-layering scheme blurs the boundaries between applications, causing you to feel as if you are interacting with the system as a whole, rather than with individual applications. (By the way, clicking the application's icon on the Dock will bring all of the application's windows to the foreground.)

1.3.5 Dialog boxes

Past Macintosh operating systems used two main types of dialog boxes: modal and modeless. A *modal* dialog box forces you to work within the mode of the dialog box only; once the dialog box is open, the only way to interact with another part of the system is to close the dialog box. Conversely, a *modeless* dialog box does not force you to interact only with it; you can simultaneously use the modeless dialog box and other parts of the system.

---

[1] Bruce Tognazzini, a noted expert on user interfaces design, has written an interesting column called "Top 10 Reasons the Apple Dock Sucks" that discusses his objections to the Dock. Check it out at http://www.asktog.com/columns/044top10docksucks.html.

Figure 1.4
Mac OS X Sheets seem fixed, or attached, to an application's document or window. They simplify identifying the owner of the Sheet.

A *Sheet* is a Mac OS X implementation of a modal dialog box. When an application displays a Sheet, it appears attached to the application's document or window (see figure 1.4). Because it attaches to its creator, you can always tell what program element the Sheet belongs to. See the Aqua Human Interface guidelines for more information about Sheets (http://developer.apple.com/techpubs/macosx/Essentials/AquaHIGuidelines/AHIGDialogs/index.html).

1.3.6 Drawers

Drawers are child windows that appear to slide out from their parent. This is another interface element that permits you to access frequently used application features or information without requiring the application to display the Drawers throughout the life of the application. To see Drawers in action, open the Mail application (located in /Applications) and click the Mailbox icon. The mailboxes for your mail accounts will slide in and out from the parent window as you click the icon (see figure 1.5).

1.3.7 Keyboard navigation

The Macintosh has traditionally been a point-and-click interface: users interact with the system using a mouse. Over the years, the system has included increasing support for system navigation through the keyboard at both the Finder and application levels. Aqua carries on this tradition by providing more keyboard options you can use to navigate the system.

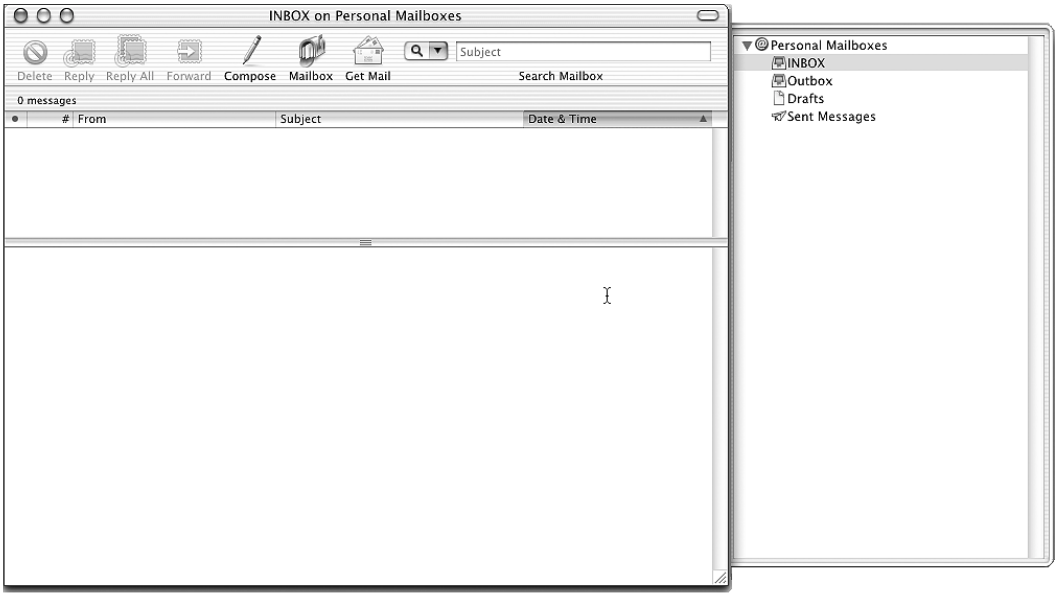

Figure 1.5 Drawers slide out from their parent window, enabling access to frequently used application features or information.

To take full advantage of the keyboard, open the System Preference program, select the Keyboard pane, select the Full Keyboard Access tab, and make sure the Turn On Full Keyboard Access checkbox is checked. The Use Control With menu enables you to change the keys associated with each command. Now, you can use the keyboard to select interface elements such as application menus and the Dock.

1.3.8 Other interface features

Mac OS X includes lots of other interface features, including transparent windows and menus that let you see through a window or menu to what is behind it. The appearance of icons and lists has improved, and there's a new help system and a new system font.

1.4 The Mac OS X architecture

From a user's point of view, the Mac OS X system is its user interface, applications, and services. For developers, however, the interface is simply a facade; behind it exists the Mac OS X operating system, a complex web of software that handles the interactions between user requests and computing resources.

The heart of this system software is the *kernel*. The kernel provides the operating system's basic computing services such as interrupt handling, processor and memory management, and process scheduling. Two types of kernels form the basis for most operating systems: the *monolithic* kernel and the *microkernel*. A monolithic kernel encapsulates nearly all the operating system layers within one program, which runs in kernel space. A microkernel implements a subset of operating system services, runs in kernel space, and is much smaller than the monolithic kernel. Additional services, implemented on top of the kernel as user programs (running in user space), export well-defined interfaces and communication semantics. To perform a service that resides outside of kernel space, the kernel communicates with the user-level service through message passing. Generally, a monolithic kernel is faster but larger than a microkernel.

The original Mac OS was more a collection of cooperating system services, whose design did not divide neatly into user and kernel domains. In addition, its handling of critical operating system tasks such as memory management and process management was showing its age, which led Apple to look into alternatives for its future OS. For example, most of us are familiar with operating systems that use preemptive multitasking and fixed-process scheduling policies. Under UNIX, one policy is for the process scheduler to divide CPU time into time slices, assigning each process a quantum of CPU time. If the running process has not terminated by the end of its quantum, the operating system will switch processes by preempting the running process and activating the next.

Contrast this to Mac OS, which implemented a scheduling called *cooperative multitasking*. It works as follows: when you run a program, the operating system loads the program into memory, schedules it for execution on the CPU, and runs the program only when the currently running program surrenders the CPU. It is the responsibility of each program, not the operating system, to occasionally hand over the CPU to allow other programs to run. As you can imagine, this scheduling is suboptimal, because one rogue program can monopolize the CPU and disallow others from running. Mac OS X is built on UNIX, and therefore uses preemptive multitasking; the kernel manages process-scheduling policies.

Another difference between Mac OS X and earlier Macintosh systems is memory management. Mac OS did not enforce memory protection of the system or application partitions. Applications were free to write to memory outside their own address space and could potentially take down other applications, as well as the entire system. Under Mac OS X, this is not possible: accessing memory outside a program's address space will result in a segment fault and the process will dump core, but it will not take down the operating system or other processes with it.

1.4.1 Architecture layers

The Mac OS X architecture is composed of several layers, each responsible for different system services. It's important to keep in mind that Mac OS X is built on top of a UNIX-based kernel, which provides the system with its plumbing (core services) and supports the various application layers with which the user interacts. It's useful to view Mac OS X as two systems, one built on the other (see figure 1.6).

At the core of Mac OS X is Darwin, an open source operating system based on Mach 3.0 and 4.4BSD. Darwin is a complete operating system that does not require higher-level Macintosh components to run. The Darwin system has two overall components: the kernel environment and the BSD emulation layer. The kernel environment provides core operating system services; the emulation layer supplies the system with the BSD user environment, or operating system personality. In fact, you can install Darwin on a PowerPC or x86 machine and use it as a stand-alone BSD-like system.

Macintosh-specific system components, built on top of the Darwin kernel environment, give Mac OS X its Macintosh character and services. Think of Darwin

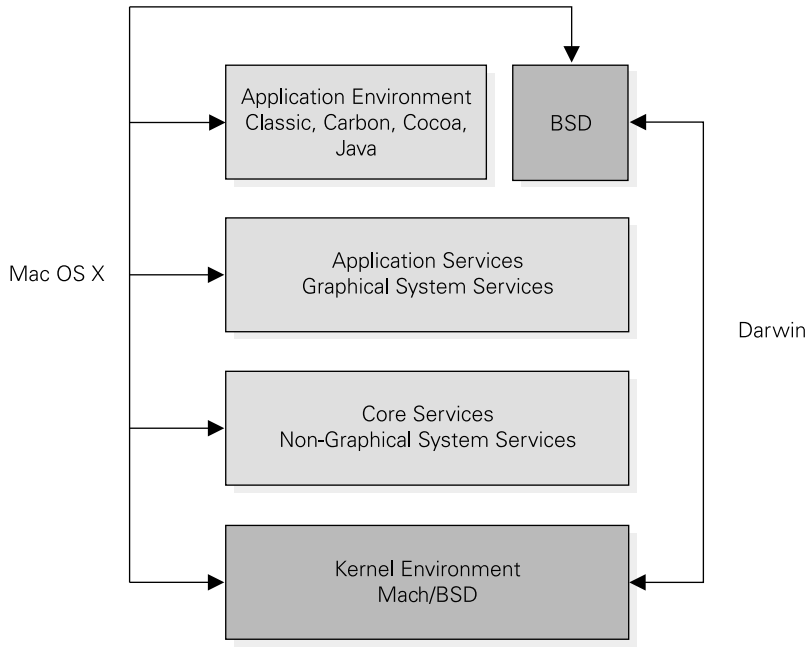

Figure 1.6 Mac OS X is a series of software layers, each providing services for the layer above it.

as the BSD-based operating system core and the Macintosh components as putting the Mac into OS X. This classification enables you to see that Mac OS X is built on top of Darwin, and that Darwin is a complete UNIX system within itself.

Let's begin with a brief overview of the Mac OS X system components:

- The lowest layer is the Mach/BSD-based kernel, called the *kernel environment*. It provides the system with core operating system services such as processor and memory management, file systems, networking, and device access and control.
- The *Core Services layer* implements a central set of non-graphical routines that various Macintosh APIs access. This layer includes facilities for application interaction with file systems, threads, and memory, and provides routines for manipulating strings, accessing local and remote resources through URLs, and XML parsing.
- Above the Core Services layer is the *Application Services layer*. Application services supply programs running within the application environment (except BSD) with user interface, windowing, and graphical support, including support for drawing graphical elements on the display, event handling, printing, and window management. This layer includes the Mac OS X window manager.
- The *Application Environment,* like Applications Services, is composed of the different application environments that give the system its user-level environment. Currently, Carbon, Cocoa, Classic, Java, and BSD form this layer, each as a separate application environment. Each provides a distinct runtime environment in which to run programs and interact with the lower layers of the operating system. For example, when you run a Mac OS X Cocoa program, you are in the Cocoa application environment; when you run a Mac OS program, you are interacting with the Classic application environment.
- Above the application environment is *Aqua*, the Mac OS X user interface. Aqua gives the Mac OS X system and programs their look and feel.

Now, let's look at each system layer and its components in more detail.

1.4.2 The kernel environment

The kernel environment supplies Mac OS X with its core operating system services. This layer is composed of two sublayers: the Mach kernel and the BSD layer, which encloses Mach (see figure 1.7). Within these layers are five primary components: Mach, the I/O Kit, BSD, the file system, and networking.

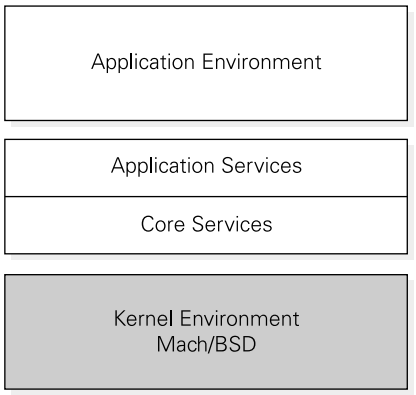

Figure 1.7
The Mac OS X kernel environment supplies the system with its core operating system services.

Mach

At its core, Mac OS X uses the Mach 3.0 microkernel (Mach 3.0 + OSF/Apple enhancements). The Mach portion of the kernel environment is responsible for managing the processor and memory (including virtual memory and memory protection), preemptive multitasking, and handling messaging between operating system layers. Mach also controls and mediates access to the low-level computing resources. It performs the following tasks:

- Provides IPC infrastructure and policies (through ports and port rights), as well as methods (message queues, RPCs, and locks) enabling operating system layers to communicate
- Manages the processor by scheduling the execution and preemption of threads that make up a task
- Supports SMP (symmetric multiprocessing)
- Handles low-level memory management issues, including virtual memory

Keep in mind that Mach is policy neutral, meaning that it has no knowledge of things like file systems, networking, and operating system personalities.

Historically, Mach implements a very small set of core system services in the kernel address space, communicating with additional services in user space through well-defined interfaces and communication semantics. The kernel implementation for Darwin integrates many of these user-space services into the kernel space.

There is a fundamental difference between how a UNIX monolithic kernel and Mach kernel use and implement processes and threads. In a UNIX kernel, the basic level of scheduling is the process, not the thread. All threads within the process are bound by the scheduling priority of the process and are not seen by

the kernel as schedulable entities. For example, if the operating system suspends a process, all its threads are also suspended.

Contrast this with Mach. Mach divides the concept of a UNIX process into two components: a task and a thread. A task contains the program's execution environment (system resources minus control flow) and its threads. With Mach, the thread is the basic unit of scheduling, as opposed to a UNIX process, which uses the process as the scheduling unit. Under Mach, scheduling priority is handled on a per-thread basis: the operating system coordinates and schedules threads from one or many tasks, not on a per-process level.

I/O Kit

The I/O Kit is an object-oriented framework for developing Mac OS X drivers, implemented in a subset of C++. Developing device drivers is a specialized task, requiring detailed knowledge, experience, and highly specific code. The I/O Kit attempts to increase code reuse and reduce the learning curve of driver development by providing programmers with a framework that encapsulates basic device driver functionality in base classes, which are extended to implement specific device drivers. Conceptually, this approach is very similar to application frameworks and class libraries. The I/O Kit infrastructure enables true plug and play, as well as dynamically loaded and unloaded drivers and dynamic device management.

BSD

Another component of the Darwin kernel environment is its implementation of BSD, which is based on 4.4BSD. The BSD kernel component sits on top of the modified Mach kernel, running in the kernel's address space. This component provides networking services, file systems, security policies, the application process model (process management and signals), the FreeBSD kernel API, and the POSIX API for supporting user space applications. It also provides applications with the BSD interface into the core services of the OS by wrapping the Mach primitives.

The traditional, or pure, microkernel design places many of these BSD components (such as file systems and networking) within user space, not kernel space. Darwin is not a pure microkernel. To address performance concerns, designers modified the kernel by placing some BSD system modules within the kernel space, traditionally reserved for Mach.

File system

Darwin's file system infrastructure is based on an enhanced virtual file system (VFS) and includes support for HFS (hierarchical file system), HFS+ (hierarchical file system plus) , UFS (UNIX file system) , NFS (network file system) , and ISO 9660.

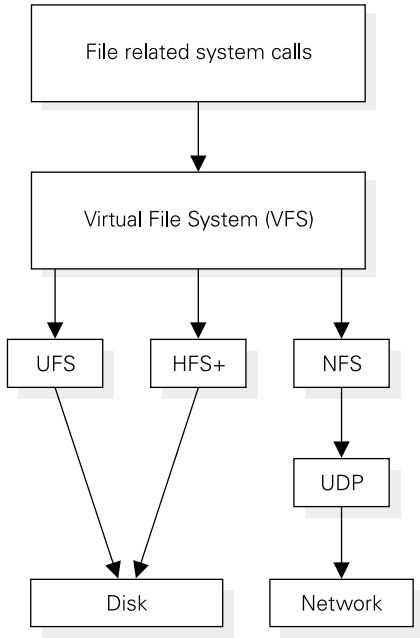

Figure 1.8
The Darwin kernel implements a Virtual File System (VFS) that translates a file-related system call into the matching call for the appropriate file system.

VFS is a kernel-level component that provides an abstract view of the physical file systems through a common interface. VFS accepts file-related system calls (open, close, read, and write) and translates them into the appropriate calls for the target file system (see figure 1.8). VFS is often referred to as supporting *stacks* of file systems (stackable), because it can interact with and add many kinds of file systems and supports augmenting existing file systems with custom code that supplies various services (such as encryption or mirroring).

Networking

Darwin's networking infrastructure is based on 4.4BSD. It includes all the features you'd expect from a BSD-derived system, such as routing, the TCP/IP stack, and BSD-style sockets. This component lives in the BSD layer of the kernel.

Kernel Extensions (KEXTs) and Network Kernel Extensions (NKEs)

Kernel Extensions (KEXTs) give developers the ability to access internal kernel data structures and add functionality to the kernel. KEXTs are dynamically loaded into kernel space without recompiling or relinking the kernel. Because KEXTs run within the kernel, a misbehaving module can potentially bring the system to its knees.

Network Kernel Extensions (NKEs) are a special instance of KEXTs. They permit developers to hook into the networking layers of the kernel and implement new features or modify existing functionality. Like KEXTs, they are dynamically loaded into kernel space and do not require recompiling or relinking of the kernel to execute.

Collectively, these components provide the core services for Darwin, and by extension, Mac OS X. A complete Darwin system adds a BSD emulation or application environment on top of this core layer, providing the userland commands and execution environment you are accustomed to in a BSD system. A complete Darwin system (core layer and BSD application environment) is a BSD-based UNIX implementation that is more than capable of running as a stand-alone operating system. You can run Darwin on a PowerPC or x86 compatible system and install it from either source code or a binary.

NOTE Remember, Darwin is an open-source project, and it is being actively developed; all source code for the operating system is available at no charge. Apple also supports several mailing lists devoted to Darwin development issues.

1.4.3 Core Services layer

The Core Services layer sits above the kernel and is responsible for non-graphical system services (see figure 1.9). Common operations are not coded into each Macintosh API (Carbon and Cocoa); instead, the Core Services layer implements a single code base that the various Macintosh APIs access. Developers use the Carbon and Cocoa APIs to construct Macintosh applications. These services are implemented in the following components:

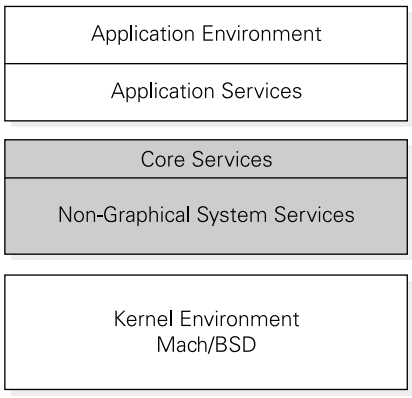

Figure 1.9
The Core Services layer (the software layer above the kernel) provides common, non-graphical routines for the Macintosh APIs (Carbon and Cocoa).

- *Carbon Managers*—A set of services, grouped under various managers, that implement routines providing applications with access to system resources and services. Managers exist for file manipulation (File Manager), text operations (Text Encoding Conversion Manager), memory management (Memory Management Utilities), and thread operations (Thread Manager). For example, when an application requires memory services, it calls a memory allocation routine located in the memory manager; this routine subsequently invokes the kernel-level system calls to manage the actual memory allocation.
- *Core Foundation*—A library that provides many low-level system services such as internationalization, string preferences, and XML services. A handy feature of the Core Foundation is its XML facilities, which include a full-fledged XML parser that implements both tree (DOM) and callback (SAX) based XML parsing.
- *Open Transport*—A single set of routines that offer transport independence and that access the underlying network protocols. Application programs interact with Open Transport through its API to perform network operations such as connecting to and receiving data from other machines. Open Transport uses the networking primitives supplied by the BSD kernel environment code.

1.4.4 Application Services layer

The next layer, called Application Services, supplies the system with the graphical services to construct user interfaces and windowed environments, as well as perform drawing operations, printing, and low-level event forwarding (see figure 1.10).

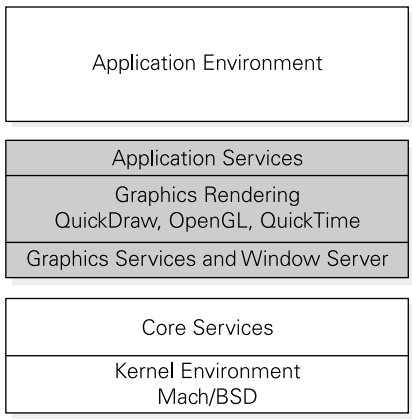

Figure 1.10
The Application Services layer supplies Mac OS X applications with graphics routines and graphics rendering using QuickDraw, OpenGL, and QuickTime.

The main component of this layer is Quartz. The term *Quartz* collectively defines the primary display technologies for Mac OS X. Quartz is composed of two layers: the core graphics services and the rendering libraries.

Core graphics services implement the Mac OS X window server and provide window management as well as event- and cursor-handling services. This sublayer does not actually render objects; the graphics-rendering sublayer that sits on top of the core services contains the following rendering libraries, which perform the graphic-rendering operations:

- *Core Graphics Rendering library*—Performs two-dimensional operations. The Core Graphics Rendering library is used for drawing and rendering using the PDF path (vector) based drawing model.
- *QuickDraw*—Performs two-dimensional operations. QuickDraw is the fundamental graphics display system for the traditional Macintosh OS; it is used to perform traditional Macintosh graphic operations.
- *OpenGL*—Renders three-dimensional operations.
- *QuickTime*—Renders multimedia and digital video in many encoding formats.
- *PDF*—(Developed by Adobe Systems.) Specifies a file format whose files are sharable across platforms. Because the Core Graphics Rendering library uses PDF for vector graphics representation, Mac OS X programs can output files in PDF format—users don't need to buy and install Adobe Acrobat. The printing system is based on this rendering model, as well.

Building these technologies into the Application Services layer provides applications with strong graphics support at the operating system level.

1.4.5 Application Environment layer

Next in this architecture is the Application Environment layer, which provides Mac OS X users with a setting in which to build and run applications (see figure 1.11). This layer, sometimes referred to as the Software Emulation layer, typically contains application emulation environments for implementations of various operating systems. In fact, you can emulate almost any operation system at this layer, including Solaris, Windows, or MS-DOS. Currently, five application environments ship with Mac OS X: Classic, Carbon, Cocoa, Java, and BSD.

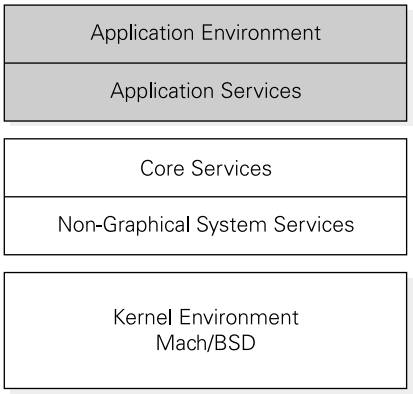

Figure 1.11
The application environment provides a setting for users to run programs. Mac OS X ships with Classic, Carbon, Cocoa, Java, and the BSD application environments.

Classic

The Classic application environment provides a setting for running programs written for Mac OS 9 and earlier. Because Apple does not endorse developing new applications for Mac OS 9, this mode's primary purpose is to support running legacy Macintosh programs. To use Classic mode, your machine must have Mac OS 9.1 or greater installed, which is the default on a typical Mac OS X machine. Therefore, a conventional Mac OS X machine will have both Mac OS X and Mac OS 9.1 installed (under Jaguar, it's version 9.2.2).

There are various approaches to running more than one operating system on a single machine. One method involves setting up a *dual boot* machine. To set up a dual boot machine, you install different operating systems on a single machine and choose the operating system you wish to run at system startup. This method is popular among users of Intel-based UNIX distributions, and it is required to run Linux/BSD and Windows on a single machine.

Another approach is *software emulation*. In this case, you run a software emulator under the host operating system that translates calls of the emulated operating system into the language of the host. This technique permits you to run different operating systems on your machine as long as you have the appropriate emulator. For example, on the Macintosh, a product called Virtual PC (http://www.connectix.com/index_mac.html) enables you to run the Windows operating system and software on your Macintosh. In addition, MacMAME (Multi-Arcade Machine Emulator) is an arcade emulator that lets you run and play your older arcade games on your Macintosh (http://emulation.net/mame).

Under Mac OS X, Classic mode is not emulated as described so far, because Classic instructions are not translated. As Sánchez pointed out:

> The Classic environment in Mac OS X creates a virtual machine inside of Mac OS X, which boots a largely unmodified version of Mac OS 9. Applications that are built for Mac OS 9 and have not been "Carbonized" run in this environment. The Classic environment replaces the hardware abstraction layer in Mac OS 9 with a series of shims that pass requests to parts of Mac OS X. For example, a memory request in Mac OS 9 is fulfilled by a memory request in the Darwin kernel. Mac OS 9 can thereby use resources managed by Mac OS X.[2]

Carbon

Carbon is a set of APIs developers can use to write applications that run under both Mac OS X and early versions of the Mac OS. The original intent of Carbon was to help developers move existing applications from Mac OS to Mac OS X.

Developers write Carbon applications in C and C++. Once an application is "Carbonized," you can run the same binary on your Mac OS X machine as on machines running Mac OS 8.1 or later.

The current Carbon API is a redesigned version of the Mac OS Toolbox. This Toolbox, originally located in the ROM and later in a file loaded by the boot loader in pre-Mac OS X systems, is a set of functions that programs access to construct the graphical elements of a program and interact with core system components. The Toolbox gave the Mac OS its unique appearance and feel, and was a fundamental element of all Macintosh programming. The Carbon API adds many new features to support the architectural changes imposed by Mac OS X. In addition, the API is much smaller, because its designers removed many Mac OS API calls.

Cocoa

Cocoa is an object-oriented environment for developing native Mac OS X applications. Cocoa provides developers with a complete component framework that greatly simplifies and facilitates the development of Mac OS X applications. Apple recommends that developers use Cocoa when writing new applications for Mac OS X.

The etymology of Cocoa begins with NeXT computer and its NeXTSTEP operating system. NeXTSTEP shipped with a set of tools and libraries called *frameworks*

---

[2] Wilfredo Sánchez, "The Challenges of Integrating the Unix and Mac OS Environments" (paper presented at the USENIX 2000 Annual Technical Conference, Invited Talks, San Diego, June 19, 2000), http://www.mit.edu/people/wsanchez/papers/USENIX_2000.

for application development. These NeXTSTEP development tools were subsequently called OpenStep, and are now called Cocoa.

Cocoa applications are currently written in one of two languages: Java and Objective-C. This may seem strange to UNIX developers who are used to developing code in languages such as C, C++, Perl, Python, and Ruby; some may even consider this limitation a reason not to develop Cocoa applications. Resist this temptation. True, many of us would prefer to use Perl or C++ as our main development language when building Cocoa applications, but any programmer who is comfortable with C or C++ can easily get the basics of Objective-C in a few days and be writing useful application in a few weeks.

In addition, some projects are attempting to bring other languages to Cocoa, including Perl, Python, and Ruby. It may just be a matter of time before your favorite language meets Cocoa.[3]

Java

The Java application environment enables development and execution of Java programs and applets. This environment supports the most recent Java Development Kit (JDK) and virtual machine, so programs developed within this environment are portable to virtual machines running on other systems. You can use Java to write applications and applets as well as Cocoa-based applications, although Objective-C is the language of choice for Cocoa development. Apple has made a strong commitment to Java on the Macintosh, so Java developers can rest assured that Java implementations and tools will be available under Mac OS X for years to come.

BSD

The BSD command environment enables users to interact with the system as a BSD workstation, typically through the Terminal application; functionally a shell. This environment supports the BSD tool set, commands, and utilities, and cumulatively provides users with a BSD-derived environment. In fact, the BSD environment and kernel environment form the complete Darwin system. This application environment enables traditional UNIX developers and users to make a smooth transition to the Mac OS X environment by providing them with the accustomed shell, tools, and command set. I for one spend most of my time in the Terminal application using Mac OS X as a BSD-based workstation.

---

[3] The PyObjC project has released a version that enables Python developers to talk to Objective-C objects from Python (http://sourceforge.net/projects/pyobjc/). See chapter 8 for more details.

1.4.6 Aqua

The top layer of the Mac OS X architecture is the Aqua user interface. Aqua is a combination interface implementation and specification that defines recommended user-interface design practices for Mac OS X applications. Think of Aqua as providing guidelines for how applications should look and behave within Mac OS X. These guidelines, documented in the Aqua Human Interface Guidelines, tell developers how to construct a Mac OS X user interface, including the proper layout of dialog boxes and window items' menu structures.

1.5 Summary

You now have a basic understanding of Macintosh user interface principles, as well as Mac OS X's user interface and design. As you can imagine, this chapter is just the tip of the iceberg. If you are interested in this aspect of the Mac OS X system, I encourage you to look at the references in the "Resources" section at the back of this book, and to explore the many online and printed sources that exist on this topic.

In chapter 2, you will learn more about the UNIX side of Mac OS X. You'll see how to accomplish common UNIX tasks under both the Mac OS X command-line interface and the Aqua interface.

2 Navigating and using Mac OS X

- The Mac OS X Terminal
- Creating user accounts
- Process management
- AppleScript and scripting languages
- Installing and running X Window under Mac OS X

Everywhere is walking distance if you have the time.

—Steven Wright

Many UNIX developers like user interfaces that are pretty minimal. Give them a simple, customizable window manager; a shell; pine for email; and programs and development tools with text-based interfaces, and they feel right at home. However, in the past few years, many members of the UNIX community have given increasing attention to developing more complete GUIs, or *desktops*, for UNIX systems. The developers of desktop environments such as GNOME (http://www.gnome.org) and KDE (http://www.kde.org) are attempting to lower the UNIX usability bar by making the system more approachable and easier to use and understand.

This chapter is about navigating the Mac OS X system and user interface and discovering the features they offer. It leverages your existing UNIX knowledge by concentrating on the commonalities between the UNIX tools and services and how they are implemented under Mac OS X. Being able to map your UNIX knowledge to Mac OS X will let you make the transition to Mac OS X more easily and quickly. The chapter concludes with information about setting up an X Window server under Mac OS X.

2.1 Introduction

To many UNIX users, a GUI is not the optimal way to interact with a system. For example, if you're applying a filter to a set of files, then using pipes and small command-line tools is much more efficient and extendable than using a GUI program. However, sometimes a GUI is preferable. If you use a command-line tool infrequently, it can be difficult to remember its options and features or recall the correct command-line syntax for a task. A GUI, on the other hand, can help by presenting the program options in a visual layout, even enabling you to save common settings for later use.

In general, desktop environments have been good for users. However, with so many competing desktops and windowing environments, UNIX systems do not provide users with a single common interface like commercial systems, such as Windows or the Macintosh. For users, this lack of consistency means relearning environments when switching between machines running different desktops. It can also complicate the work of developers building applications for UNIX systems and sometimes force them to target a particular environment.

With the introduction of Mac OS X, users now have a UNIX-based system with a single, well-thought-out interface—one designed for usability. UNIX developers coming to the platform should take a serious look at the interface, and will most likely find it very useful in maneuvering through the system and accomplishing development tasks.

2.2 Shells

Mac OS X supports interacting with its BSD underpinnings through a program called the Terminal, located in /Applications/Utilities. The Terminal application implements a command interpreter, or *shell* (see figure 2.1). The shell's basic function is to accept user commands (in the command language of the shell), parse them, and pass them to the operating system for execution.

For many Macintosh users, interacting with the computer using a command shell is enough to make them run and hide. Remember, the Macintosh and UNIX operating systems have different design goals and user cultures: the designers of the Macintosh built the system as a single-user personal computer with the goals of simplicity and ease of use. The system should empower normal people to use computers, and not require them to be programmers or system administrators.

On the other hand, we can trace the origin of UNIX to the time-sharing systems proposed and developed at MIT in the mid-1950s through 1960s—most notably CTSS (Compatible Timesharing System; http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?CTSS) and MULTICS (MULTiplexed Information and Computing Service; http://www.multicians.org). Time-sharing enables multiple users to simultaneously access computing resources. Once time-sharing was established, a shift occurred in the way people viewed and used large-scale computers. Rather than considering computers as calculating machines that processed jobs sequentially,

Figure 2.1
The Terminal program functions as an xterm in the Mac OS X environment, which you can customize to use different shells.

users began to view them as machines embodying interactive properties; many users could concurrently share a single machine's computing resources. The shell was a natural outgrowth of time-sharing—users needed an interactive, extendable method of communicating with the computer. (Louis Pouzin, then a staff member of the MIT computing center, first introduced the concept of the modern shell in 1963.)

MULTICS was one of the most innovative time-sharing systems of its day and had many contributors, including MIT, GE, and Bell Laboratories. The Bell Labs group included Ken Thompson, Dennis Ritchie, M. D. McIlroy, and Joe Ossanna. In 1969, Bell Labs pulled out of the MULTICS project, and Ken Thompson began work on a new operating system. The new system was directly influenced by MULTICS, and soon grew into UNIX.[1]

From its beginning, UNIX was a multiuser system that facilitated the sharing of computer resources among many users. Most UNIX users were, and still are, system hackers and programmers who love the power and possibilities of the system.

In Mac OS X, two vastly different system design histories and user cultures converge. This situation naturally presented the designers of Mac OS X with quite a few decisions. For example, as I stated earlier, the very idea of using a shell is contrary to the design goals of the Macintosh. However, Mac OS X is a different beast and is built on UNIX, so it is natural to include a command shell for interacting with the system. Apple has done its best to hide the shell from the average Macintosh user, but experienced UNIX users will look for this program first and gravitate to it instantly.

The Terminal program supports many shells, which you can customize through the program's preferences dialog box and initialization files. By default, the system comes with the following shells, contained in the /etc/shells file:

```
% cat /etc/shells
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
/bin/csh
/bin/sh
/bin/tcsh
/bin/zsh
```

You can change the shell that Terminal uses by selecting Terminal→Preferences and clicking the Shell option in the Preferences dialog box (see figure 2.2). Once you choose a shell, you can customize it in the usual manner. For example, I use

[1] See appendix D for more detailed information on the etymology of UNIX.

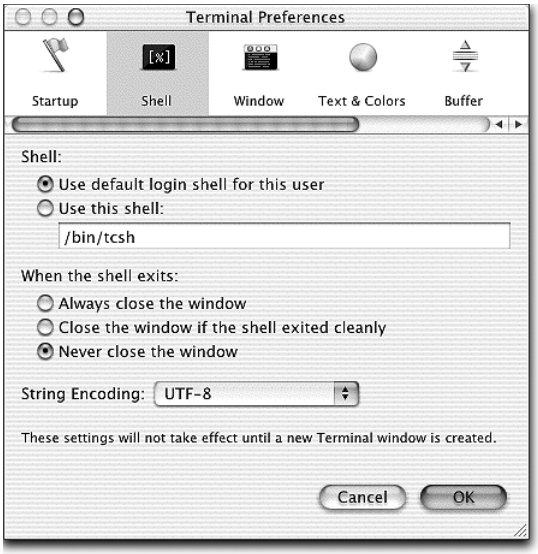

Figure 2.2
You can use the Terminal program's Preferences dialog box to select many customization options, including your active shell.

tcsh, an enhanced version of the Berkeley UNIX C shell (csh). I copied the .cshrc file from my Solaris box to my home directory on my Mac OS X machine and modified it for the new system.

2.2.1 Terminal features

Regardless of the shell you choose, some features of the Terminal program are common to all shells. One interesting set of features helps bridge the gap between the command line and Finder interfaces.

Imagine you have a Finder window open and wish to change to this directory in the Terminal program. Open the shell and type `cd` (change directory), followed by a space, at the prompt. Next, drag the folder from the Finder window into the Terminal window. Doing so will copy the absolute path of the directory to the prompt (see figure 2.3). This feature is especially useful when you're dealing with long directory paths or directories that are highly nested. In addition to copying directory paths, you can use this technique to copy file paths by dragging a file from a Finder window into the Terminal window.

From the Terminal program, you can open directories, files, and programs within the Finder using the *open* command. For example, typing `open` followed by a directory name opens the specified directory in a Finder window; typing `open` followed by a filename opens the file. If you type `open .cshrc`, the file .cshrc is opened in TextEdit (a Mac OS X text-file editor).

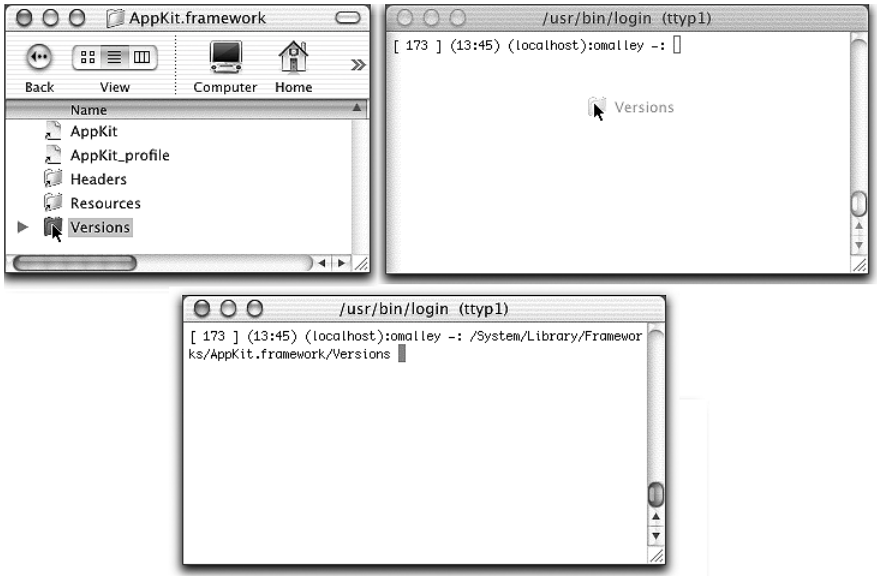

Figure 2.3 Dragging a folder to the Terminal window is an easy way to copy long path names with no typing.

2.3 Help system

UNIX traditionally uses manual pages, or man pages, to document commands and tools. To view man pages, set the man path environment variable (`MANPATH`) to the location of your system's man pages, and the `PAGER` variable to the program you want to filter the pages (typically, `more` or `less`). In practice, the `PAGER` variable lets you specify the command or program that will display the man page. For example, if you set `PAGER` to `emacs`, the system will display man pages within the `emacs` editor. If you set it to the `cat` command (which writes a file to standard output), the system will send the man page to the `cat` command. The `more` and `less` commands are common choices, because they enable you to view a man page one screen at a time.

The `man` program takes one argument (the command name to look up); it finds the corresponding documentation file, runs the file through `nroff`, and pipes its output through the `more` command (`nroff`, and its supporting utilities, are used to format text files). As you would expect, the `man` command is available under Mac OS X for getting help on UNIX commands.

2.3.1 Help Viewer

To get help on Mac OS X applications, you use the Apple Help Viewer (see figure 2.4), which you access from a Mac OS X program's Help menu. Mac OS X programs implement the Help menu as the rightmost Application menu and use the help system to present program information to the user. Most Mac OS X programs (Cocoa, Carbon, and Java) provide help in this manner, rather then using man pages. This is true of any GUI program written for Mac OS X, as well as GUI programs included with the OS.

Figure 2.4
Mac OS X applications include online help through Apple Help Viewer.

2.4 User accounts and privileges

On a UNIX system, there are two types of users: those with root privileges and those without. By going root, you have full access to every aspect of a UNIX system and can roam the system at will, installing software in privileged locations, updating system configuration files, and deleting any file you wish. Basically, you are free to make the system hum along—but you can easily take it to its knees with a misplaced command.

Apple recognized that a middle ground exists between user and root privileges, so it introduced new *administrator* privileges. Users with administrator privileges have all the rights of a normal user but can also install new programs, create directories outside the home directory, and add new users to the system. However, you can't do some things with administrator privileges, such as manipulate the System Folder, view the contents of another user's directory, or edit many system configuration files. For these operations, you still need root access.

Because Mac OS X is first a consumer operating system, Apple naturally discourages users from obtaining root access; toward this end, the root account is disabled, to protect inexperienced users from clobbering their system. However, if you plan to do any work that involves tuning the system, configuring system services, or general hacking, root is a must.

2.4.1 Creating user accounts

You create user accounts from the System Preference application (available from the Dock or within /Applications), using the Users pane (see figure 2.5). When you create a user, you can assign normal privileges or administrator privileges, but not root. There are two primary ways to permit root privileges under Mac OS X: by using the `sudo` ("soo-doo") command and by directly enabling the root account.

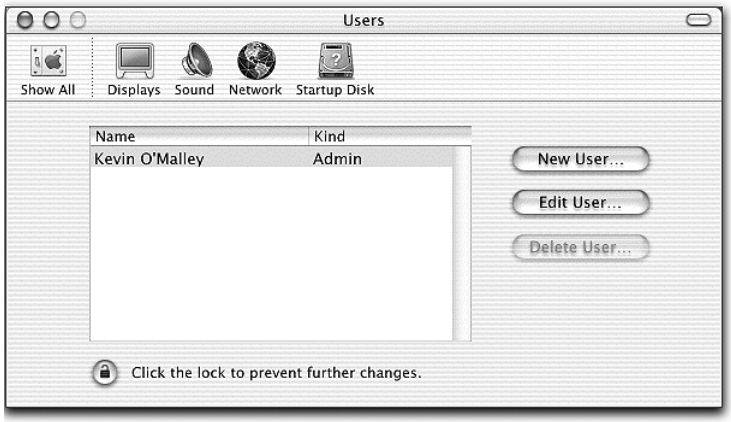

Figure 2.5 You add users to the system and assign administrator privileges using the System Preference program's Users pane.

The sudo command

The `sudo` command lets a user execute a command as root. Only certain users can use this command, and only certain commands can be run; these are defined as configuration parameters and stored in /etc/sudoers. Mac OS X installs the `sudo` program as part of the default load and permits users with administrator privileges to use the command.

You can use the command two ways. First, you can add the prefix `sudo` to the command you wish to run as root. The following example shows the result of a command run first as a regular user and then as root, using the `sudo` command:

```
% more /etc/master.passwd
/etc/master.passwd: Permission denied
% sudo more /etc/master.passwd
Password:
##
# User Database
#
# Note that this file is consulted when the system is running
# in single-user mode. At other times this information is handled
# by lookupd. By default, lookupd gets information from NetInfo,
# so this file will not be consulted
# unless you have changed lookupd's configuration.
##
nobody:*:-2:-2::0:0:Unprivileged User:/dev/null:/dev/null
root:*:0:0::0:0:System Administrator:/var/root:/bin/tcsh
daemon:*:1:1::0:0:System Services:/var/root:/dev/null
unknown:*:99:99::0:0:Unknown User:/dev/null:/dev/null
www:*:70:70::0:0:World Wide Web Server:/Library/WebServer:/dev/null
```

(In the preceding example, type your password at the password prompt.) This method enables you to run a command as root for a defined interval (usually five minutes) without retyping your password.

Second, to enable root access indefinitely, use `sudo` with the `–s` option and enter your password at the password prompt:

```
% sudo -s
Password:
```

Now, commands run under root. Typing `exit` will end the session.

Enabling the root account

You can also run commands as root by enabling the root account. To do this, you need to run the NetInfo Manager system administration tool. NetInfo (located in /Applications/Utilities) is used to perform administrative tasks on Mac OS X. The program, originally used under NeXTSTEP, is a hierarchical distributed database of system information.

To use NetInfo Manager to enable the root account on your system, follow these steps:

1. Launch the program and select Domain→Security→Authenticate. (Under Jaguar—Mac OS X 10.2—select Security→Authenticate; the program is no longer under Domain.)
2. Enter your password when prompted and click OK (remember, for this technique to work you must have administrator privileges).

3 Select Domain→Security→Enable Root User. Reauthenticate by selecting Domain→Security→Authenticate and entering your password.

To test the root account, open a shell (using the Terminal program) and substitute your user identity with root:

```
% su -
Password:
root#
```

2.5 Booting and default services

When you boot a Mac OS X system, the system first runs the *BootROM* firmware to perform a *Power On Self Test* (POST), initialize hardware, and select an operating system to use. Next, the *BootX* loader takes over and loads the operating system kernel environment from disk. Once the kernel and devices are loaded, BootX calls the kernel's initialization function and mounts the root file system. Kernel initialization includes initializing the components of the kernel environment (including data structures, Mach, BSD, and the I/O Kit) and running the `mach init` process, which enables messaging (over ports) and runs the BSD `init` process (as PID 1). The `init` process is the parent, or owner, of all subsequent processes. It performs tasks such as running the system in either single- or multiuser mode, running the initialization scripts (the `rc` scripts and SystemStarter), launching the login window process (which presents the login window and processes user login attempts), and performing cleanup tasks for child processes.

The `rc` scripts perform BSD-style startup. They start processes such as `kextd` and `update` (flushes the file system cache at regular intervals). The last task the `rc` scripts perform is to launch the `SystemStarter` process. `SystemStarter` runs the default Mac OS X startup items, located in /System/Library/StartupItems, as well as user-defined startup items (/Library/StartupItems). The `SystemStarter` process runs services such as `portmap`, `autodiskmount`, `syslog`, `DesktopDB`, `inetd`, `sendmail`, and `cron`. The services that are started are determined by the entries in the /etc/hostconfig file:

```
% cat /etc/hostconfig
##
# /etc/hostconfig
##
# This file is maintained by the system control panels
##
```

```
# Network configuration
HOSTNAME=-AUTOMATIC-
ROUTER=-AUTOMATIC-

# Services
AFPSERVER=-YES-
APPLETALK=-NO-
AUTHSERVER=-NO-
AUTOMOUNT=-YES-
CONFIGSERVER=-NO-
IPFORWARDING=-NO-
MAILSERVER=-NO-
MANAGEMENTSERVER=-NO-
NETINFOSERVER=-AUTOMATIC-
RPCSERVER=-AUTOMATIC-
NETBOOTSERVER=-NO-
NISDOMAIN=-NO-
TIMESYNC=-YES-
QTSSERVER=-NO-
SSHSERVER=-NO-
WEBSERVER=-NO-
APPLETALK_HOSTNAME="Kevin O'Malley? Computer"
COREDUMPS=-YES-
```

2.6 Programs and Mac OS X bundles

Under UNIX, you typically build and install programs from source code using the `./configure`, `make`, and `make install` commands. The `make install` command copies all program files to a default location or a location specified as a command-line option to the configure script. The downside of this approach is that program elements are not necessarily placed under a single directory and can be spread out over the system.

Many UNIX implementations also support program installation from packages using a package management tool. The advantage of this approach is that the package manager software keeps a list of all installed programs and program components. When you remove a program, all program components are also removed.

Mac OS X takes a different approach. When you install a Mac OS X program, all files that make up the program are stored under a single directory, called a *bundle*. A bundle is a directory that holds all program components in one location, including the application and application resources such as graphics and sound files. Figure 2.6 shows the contents of a bundle from the Finder and the shell. Note that double-clicking on the program icon in the Finder window will run the program and does not open the directory.

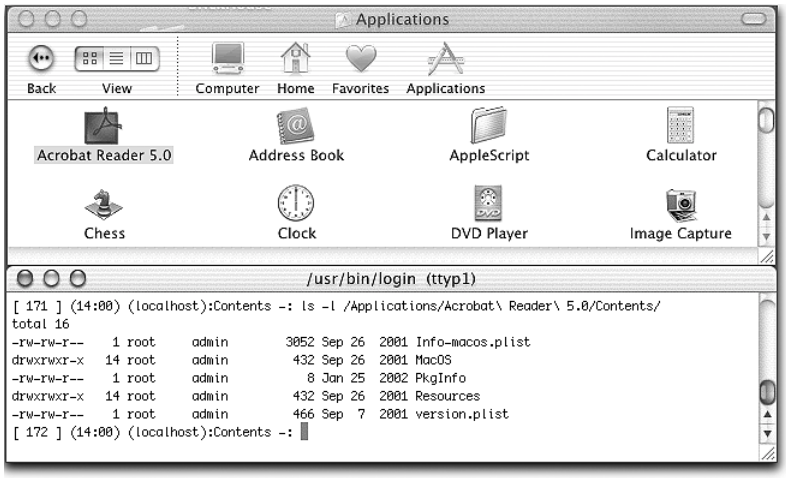

Figure 2.6 Mac OS X applications are stored on disk in bundles. Bundles group program components under a single directory.

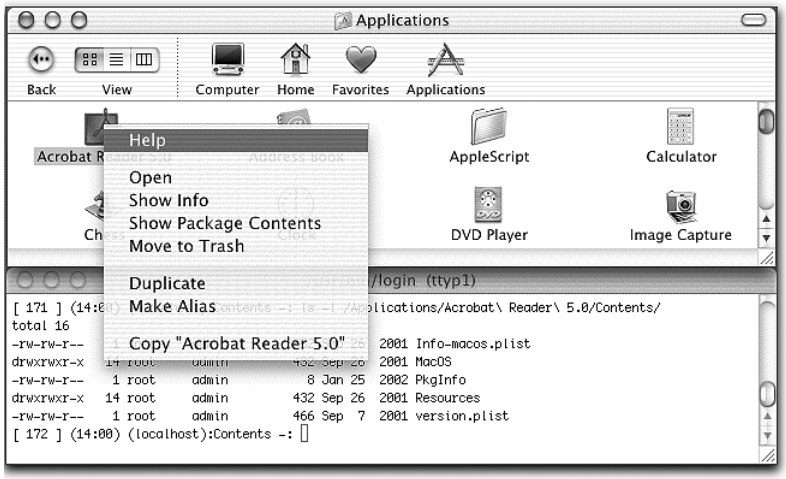

Figure 2.7 You can view the contexts of a folder from the Finder by holding the Control key and clicking on the program's icon.

You can also view the contents of the folder by holding the Control key, single-clicking on the program's icon, and selecting Show Package Contents from the pop-up menu (see figure 2.7).

Bundles offer many advantages, but the primary benefit for users is that moving a program from machine to machine, or from disk to disk, is as simple as a drag-and-drop operation. Imagine you have a collection of programs on one machine and wish to transfer them to another machine. You can simply share one of the machines and drag and drop the programs from the Finder window to the new machine—no reinstallation or configuration is required.

2.7 Security issues

In today's computing environments, networks are ubiquitous. Overall, this is a good thing; it leads to a more productive and enjoyable computing experience for users. The problem is that as soon as you put one computer online, you open it to attack from anyone who has access to the network. Users of UNIX systems are well aware of these risks and typically limit the number of services a system runs to the bare minimum (`ssh` only), as well as implementing some sort of software firewall (`ipchains` or TCP Wrapper). In practice, replacing the `telnet` and `ftp` daemons with secure shell (`ssh`) is a good step in eliminating many security risks.

Mac OS X comes with an IP firewall program called `ipfw`. Unfortunately, this command-line tool is a bit daunting to use and requires experience to configure correctly. Enter BrickHouse (http://personalpages.tds.net/~brian_hill/brickhouse.html), a program that provides a Mac OS X GUI for `ipfw` (see figure 2.8).

With BrickHouse, it's simple to set up a software firewall for your machine without getting into the gory details of `ipfw`. In fact, BrickHouse is a good example of how to construct a Mac OS X GUI application that interacts with a UNIX tool. (This technique is of great value to UNIX developers moving to Mac OS X; I cover it in detail in chapter 7.)

2.8 File system

The UNIX file system is made up of a hierarchy of files, directories, links (hard and soft), and mount points under the directory /, called root. The organization of the file system as seen from the Mac OS X Finder is somewhat different. The file system visible from the Finder is separated into four domains, each of which defines an area that holds files defined for a particular function:

- *User domain*—Holds home directories for user accounts on the system
- *Network domain*—Holds resources shared among all users that reside on the local network

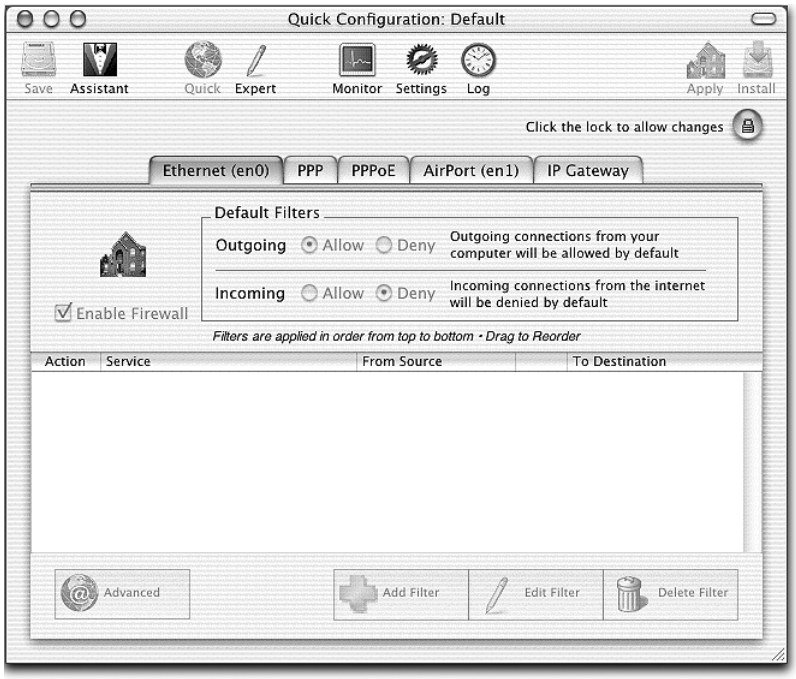

Figure 2.8 BrickHouse provides an interface for the ipfw command-line tools, giving you a simple way to set up software-based firewalling.

- *Local domain*—Holds resources shared among users, such as programs (in the Applications folder and Library files)
- *System domain*—Holds system software

Tables 2.1 through 2.3 describe the contents of each Mac OS X file system domain.

Table 2.1 Contents of the User domain

Name	Description
Desktop	User-defined desktop items (programs, aliases, documents the user has placed on the Finder desktop)
Documents	User documents
Library	Application resources
Movies	User movie files
Music	User music files, such as mp3s

Table 2.1 Contents of the User domain *(continued)*

Name	Description
Pictures	User image files
Public	Shared items enabled through the System Preference sharing option
Sites	User bookmarks for web sites and the user's web site
Applications	Private user programs

Table 2.2 Contents of the Network domain

Name	Description
Applications	Applications available to all users over the network
Library	Application and system resources for programs that reside on a network volume, which are available to all users of the system

Table 2.3 Contents of the Local and System domains

Name	Description
Applications	Default location for Mac OS X applications that are available to all users of the system
Applications/Utilities	Default location for Mac OS X administrative applications that are available to all users of the system
Library	Application and system resources that are available to all users of the system
System (both Mac OS X and Mac OS 9)	System software

2.8.1 Finder

The Mac OS X Finder presents a user-friendly view of the file system but hides many of the files and directories that are visible from the shell. Figure 2.9 shows the root file system from the shell and Finder.

The Finder view hides many of the UNIX-specific files and directories. This is intentional: UNIX is the underpinning of the system, and, for most users, seeing this information would only detract from their experience and provide little or no functionality.

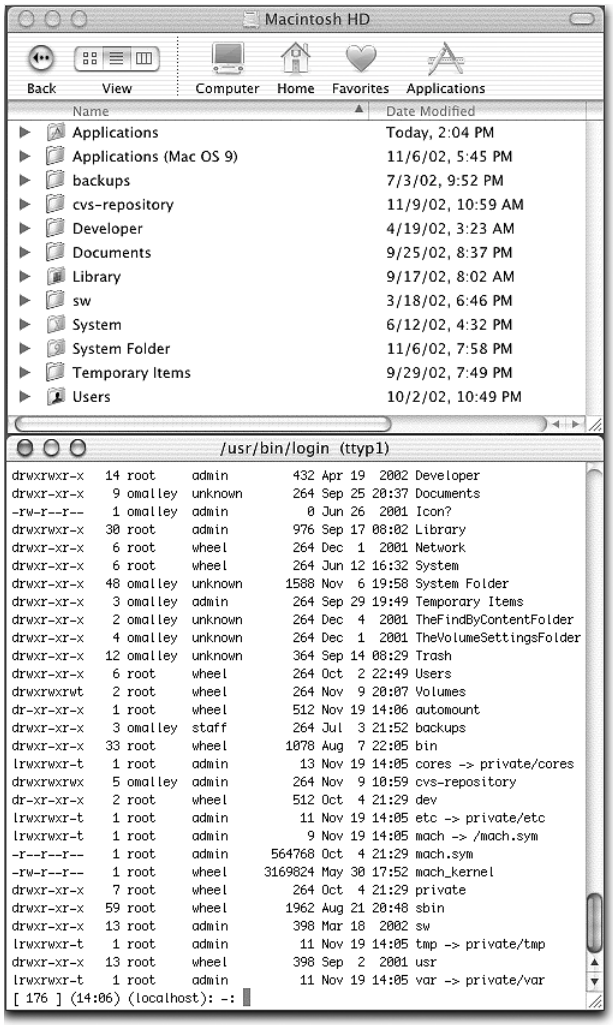

Figure 2.9
The Finder hides many of the UNIX-specific file system items from users, including files and directories.

If you wish to see these items in the Finder window, you need to edit a configuration file. The file /.hidden lists and controls the items that are not visible in the Finder:

```
% ls -l .hidden; cat .hidden
-r--r--r--    1 root     wheel         152 Sep  2  2001 .hidden
automount
bin
cores
Desktop DB
Desktop DF
Desktop Folder
```

```
dev
etc
lost+found
mach
mach_kernel
mach.sym
private
sbin
tmp
Trash
usr
var
VM Storage
Volumes
```

As you can see, user root owns this file. You can change the files that are visible from the Finder by adding or removing items from this file (as root), logging out of the current session, and logging back in. (You can also make a directory or file invisible from the Finder by prefixing its name with a period [.].)

2.8.2 Case sensitivity and pathname delimiters

The primary file system for Mac OS X is HFS+. It is case insensitive, but it maintains case information so the UNIX side of things can preserve case sensitivity.

Another feature of Mac OS X is its treatment of special characters in filenames—specifically, the pathname delimiter. The original Mac OS used a colon as a path delimiter, but UNIX has always used the forward slash (/).

Try this:

1. Create a text file, naming it test_file.txt (`echo "" > test_file.txt`).
2. From the Finder, locate the directory that holds the file and rename the file test/file.txt.
3. Go back to the shell and list the directory contexts (`ls`). You'll see that a colon has replaced the forward slash in the filename.

This example demonstrates the result of the conversion between a colon and forward slash at the VFS layer. VFS provides an abstract view of the physical file systems through a common interface. VFS accepts file-related system calls (`open`, `close`, `read`, `write`) and translates them into the appropriate calls for the target file system (see figure 2.10).

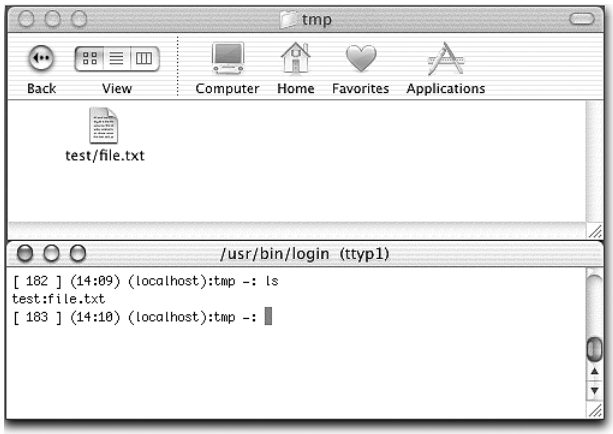

Figure 2.10
The VFS layer in the kernel is responsible for translating Mac OS X file delimiters to their UNIX equivalent.

2.9 Single-user mode

UNIX systems permit users with root privileges to boot the system under various run levels. Each run level provides the user with different functionality. For example, under Solaris, level 1 boots the system in System Administrator mode, mounting file systems, and enabling a subset of system services. Under RedHat Linux, run level 3 enables multiuser mode, 5 boots the system into X11, and 1 boots into single-user mode.

Most, if not all, UNIX distributions support single-user mode, which is primarily used for diagnostics and system maintenance. Typically, single-user mode enables a very small subset of commands that let you perform basic system maintenance operations. To boot Mac OS X into single-user mode, hold the Command+S keys at startup.

Single-user mode disables most services. In fact, the only services run are as follows:

```
/sbin/init –s
/sbin/mach_init -s
-sh (sh)
```

The `mach_init` process enables messaging over ports by bootstrapping the Mach port server and running the `init` process. Without `mach_init`, there would be no way for the kernel to communicate with other system components. During the final stage of the boot process, /sbin/init is run. The `–s` option tells the process to run the system in single-user mode. One of the operations performed by the init process is to fork a process that runs the shell `sh`.

Once in single-user mode, you can run the `fsck` command to examine and fix the boot volume's file system. To exit single-user mode, type `exit`, which continues with the boot process. Typing `reboot` will restart the system.

2.10 System log files

BSD system log files are stored in their usual BSD location: the /var directory. Mac OS X provides a GUI tool called Console, located in the Applications/Utilities folder, which displays the console.log file. In addition to the UNIX log files, two log folders contain Mac OS X–specific log files: ~/Library/Logs and /Library/Logs hold log files for disk copies and file service and directory service errors.

2.11 Processes management

UNIX users quickly become familiar with performing process management tasks through the `kill`, `top`, `nice`, `renice`, and `ps` commands. These commands enable you to control, terminate, and get information about a process by specifying a *process identifier* (PID). A process identifier is a unique integer assigned to a process by the operating system that enables the system (and you) to identify and interact with the process. Darwin, and by extension Mac OS X, supports these process-management commands through the BSD user environment.

In addition to the UNIX commands, Mac OS X contains a GUI-based process management tool called ProcessViewer that performs similar functionality. ProcessViewer, located in /Applications/Utilities, lists instantiated processes, displays information on each process, and enables you to kill a running process (see figure 2.11). The Show pop-up menu lists the categories of processes (All Processes, User Processes, Administrator Processes, and NetBoot Processes), enabling you to filter the program's ProcessViewer displays. (The program is self-explanatory; to learn more, run it and investigate its features.)

One limitation of the program is that you cannot send processes different types of signals. Imagine you wrote a program that performs an action when it gets a `SIGUSR1` signal. ProcessViewer does not permit you to send this signal to the process—it only permits you to kill a process by double-clicking on a process or name or selecting the process and pressing Command+Shift+Q. Presumably, ProcessViewer sends the process a `KILL` signal.

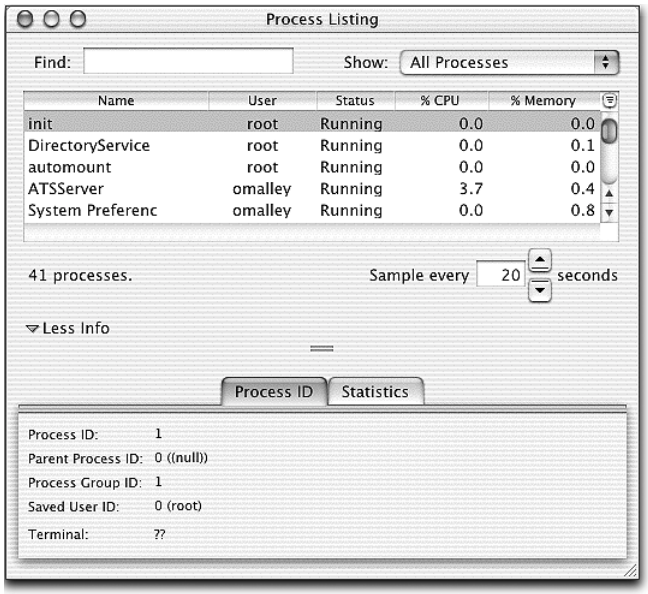

Figure 2.11
The Mac OS X ProcessViewer lists running processes and displays information about each process. It also enables you to kill an instantiated process.

2.12 Common commands and tools

Most users perform a limited number of tasks when using a computer. My day is usually spent at the command line, within `emacs`, or using a web browser. When you switch to a new operating system, it is useful to first learn how to perform common tasks on the new system so you can get right to work. For example, a common task is to search for all files greater than a certain size, say 1MB. Under UNIX, you accomplish this as follows:

```
% find / -size +1048576c -exec ls -l {} \;
```

It would be helpful to know how to perform the same task within the Mac OS X environment. Of course, you can use the same command from the shell, but you are interested in how to do this within the Macintosh environment (see appendix B for the Mac OS X GUI-based equivalent of the UNIX `find` command).

Let's look at an example of one of these mappings. Under UNIX, a common operation is to view the state of the system or a particular process using the `top` command (see figure 2.12). To get information on the top processes consuming CPU, you use the following command (by default, `top` displays updates every one second; the `–s 2` option tells it to update every two seconds):

```
% top –s 2
```

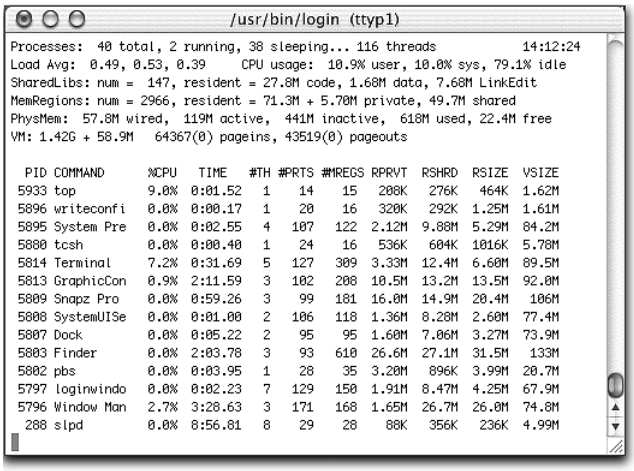

Figure 2.12
The `top` command displays an updated sample of system usage statistics.

Figure 2.12 shows the result.

As mentioned earlier, Mac OS X comes with a program called ProcessViewer that provides similar functionality but displays the information in a GUI (see figure 2.13). To use ProcessViewer, open the /Applications/Utilities folder and double-click on the ProcessViewer icon.

As you can see, much of the same information is displayed, but the GUI provides access to program features through menus.

Appendix B lists common UNIX commands and their Mac OS X equivalents. The information is not all-inclusive, but it will get you started.

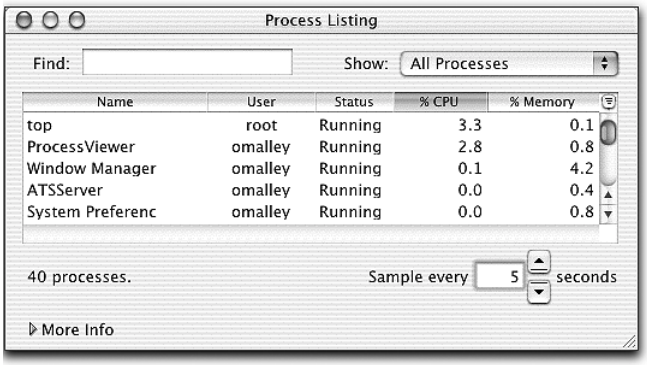

Figure 2.13
A Mac OS X GUI program (ProcessViewer) that displays information similar to that from the UNIX `top` command

2.13 Scripting languages

Historically, UNIX systems have provided strong support for text processing, filtering, and program automation through commands, pipes, shell scripts, and high-level scripting languages such as Shell, Perl, and Python. The most basic technique is to use standard UNIX commands combined with pipes.

For example, imagine you wish to count the number of lines (comments and empty lines) in a project's source tree and display the result. This task includes finding all target files under the project directory, counting the number of lines in each source file, summing the source lines, and printing the results. A common UNIX solution is to use various UNIX commands linked together with pipes.

To find the source files, you use the `find` command; to count the lines, you use the `wc` command; to construct arguments and execute a utility (`wc`), use xargs. To connect the commands, you use a *pipe* (`|`). Using these commands and a pipe, you can solve the problem without writing a single line of code:

```
% find myproject -name "*.c" | xargs wc -l
```

Another technique is to use specialized tools like `ed` and `awk` to perform tasks such as filtering lines in a set of files and extracting and formatting information. Both UNIX commands and tools such as `ed` and `awk` provide you with primitives, but they do not give you the programmatic infrastructure to perform tasks that are more complex. Enter scripting languages.

Scripting languages, such as Perl and Python, enable you to perform many of the tasks you accomplished using UNIX commands and tools, but give you plenty of infrastructure to extend and enhance your solutions. In addition, these languages let you write programs that talk over a network, provide a GUI for user interaction, and perform mathematical operations. Scripting languages are not new; they have existed since the 1960s. Early languages included JCL (Job Control Language), sh (the first shell), and Rexx; today's popular languages include Perl, Python, JavaScript, and Tcl.

2.13.1 AppleScript

All your favorite UNIX scripting languages, commands, and tools, are available under Mac OS X from the Terminal application. However, the Mac OS X offers another scripting language that is specific to the Macintosh: *AppleScript*. AppleScript, developed by Apple, is a high-level scripting language that facilitates the manipulation of application and system services. The advantage of AppleScript over other scripting languages is that AppleScript is a system- and application-level

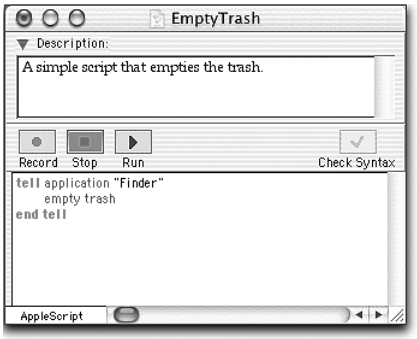

Figure 2.14
The Script Editor is the main development tool for writing and running an AppleScript.

scripting language supported by most Macintosh applications. Because support for AppleScript is built into the Macintosh operating system, there is tight integration with core system services and *Inter-Process Communication* (IPC) facilities between applications.

The main reason to use AppleScript over Perl or Python is its ability to control other programs and use their services. You can do something like this with Perl using the *open* and *system* calls, but with AppleScript the technique is far more substantial. In Perl, you call programs as black boxes; but AppleScript gives you access to the application's internals, so you can script many of the features that are available to a user interacting through the program's GUI.

AppleScript uses *AppleEvents* as its primary communication primitive, which facilitates the sharing of services between applications. AppleEvents are defined messages that enable applications to extend their functionality by using the services of other applications and share their own operations with other applications. AppleScript communicates with applications by sending AppleEvents to other AppleEvent-enabled applications or system processes to request services and receive the result of the operation.

Let's take a quick look at AppleScript and get a feel for how easy it is to write scripts. The AppleScript editor, Script Editor (see figure 2.14), is located in the /Applications/AppleScript folder and is loaded as part of the default Mac OS X installation.

You use the Script Editor as your main development environment for writing and testing scripts. AppleScript is an easy-to-understand, English-like language, structured as a series of single or compound statements.

Imagine you wish to create an AppleScript to connect to a specific host over `ssh`. To do so, follow these steps:

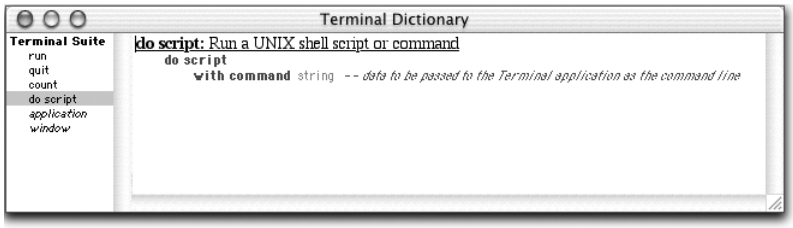

Figure 2.15 AppleScript-enabled programs like Terminal export accessible operations using dictionaries.

1. Open the Script Editor and select File→Open Dictionary. Select Terminal from the list and click the Open button.
2. The Open Dictionary menu item opens a window that lists all programs with which your script can communicate. The Terminal Dictionary window displays all commands and objects exported by the Terminal program (see figure 2.15).
3. From this list of objects and commands, you can see the aspects of the program that are scriptable. Enter the following script into the Script Editor and save the script as an application:

```
tell application "Terminal"
  run
  do script with command "ssh host.my.domain.edu"
end tell
```

4. To connect to the host, double-click on the script from the Finder, or click the Script Editor's Run button.

AppleScript is a cool and useful technology for controlling many aspects of your Mac OS X system. It takes very little time to learn, it's powerful, and it enables you to tie together the services of many Mac OS X applications to perform powerful tasks. In chapter 7, you will learn more about AppleScript and its technologies.

2.14 Development tools

The default load of Mac OS X does not contain any UNIX or Mac OS X–specific development tools, such as `gcc`, `g++`, `gdb`, `RCS`, `CVS`, Project Builder, or Interface Builder. These tools and others are available free from the Apple Developer site (http://developer.apple.com). Appendix A provides all the information you need to download and install the complete suite of Apple development tools.

2.15 X Window under Mac OS X

Mac OS X is really two systems in one: you can use it as a Macintosh system through its Aqua GUI or as a BSD box through the BSD user environment and shell (using the Terminal program). However, Terminal is text based and only supports text-based programs. The default installation of Mac OS X does not come with an X Window server, so you cannot run X11-based applications from the Terminal. Luckily, there are free X Window servers that run on Mac OS X, permitting you full access to local and remote X Window applications under Mac OS X.

In addition, many active projects are being developed to bring the full BSD tool chain to Mac OS X. These projects provide users with infrastructure that greatly simplifies locating and installing UNIX and BSD tools that do not come standard with Mac OS X. This process is exciting and is one of the primary advantages of using Mac OS X. Through the work of many individuals, most of whom are volunteers, you now have the means to replicate your UNIX work environments and tools on the Macintosh.

X Window, developed at the Massachusetts Institute of Technology (MIT), is the primary graphics display and windowing system for UNIX user interfaces. X Window lets you display basic graphic elements such as pixels, lines, and text, as well as advanced interface components like windows and buttons, on a computer terminal. UNIX window managers like `twm` (http://www.plig.org/xwinman/vtwm.html) and `fvwm` (http://www.fvwm.org), and desktops such as KDE (http://www.kde.org) and GNOME (http://www.gnome.org), use the services of X Window.

Mac OS X does not use X Window as its graphic display system. Instead, it uses its own proprietary system called Quartz to handle graphics operations. The Mac OS X user interface for Quartz is called Aqua. X Window and Quartz are two fundamentally different graphics and display technologies. For example, you cannot run X Window applications under Mac OS X, because Quartz does not support X Window. However, as I mentioned earlier, the software community has come to the rescue with freely available X Window servers for Mac OS X.[2]

[2] Projects include XFree86 on Darwin and Mac OS X (http://mrcla.com/XonX) and the XDarwin Project (http://www.xdarwin.org).

NOTE You will often see the terms *rooted* (full screen) and *rootless* in the documentation that accompanies X servers for Mac OS X. Rooted means X11 occupies the entire screen. In this mode, your display looks like an X Window session on any other UNIX machine. You can switch between the X Window and Mac OS X environments, but only one is visible at a time. Rootless mode enables both X Window and Aqua to coexist on the display simultaneously. You switch between applications in each environment by clicking on the appropriate application window.

2.15.1 Installing the X server

There are many ways to install the X server on your machine, but all share some common steps. First, you need to install the Mac OS X version of XFree86; then you install the X server software, called XDarwin. Both software packages are free. For simplicity, you will install both from a combined binary distribution.

To install the package, follow these steps:

1. Open your web browser and point it to the following location: http://www.osxgnu.org/software/Xwin/xfree86.
2. Download the XFree86 for Mac OS X Rootless version X by choosing the appropriate download site and clicking the Download button.
3. If your browser has not already done so, decompress the distribution. This step should result in an Xfree86Complete-[version].mpkg file.
4. Double-click on the file icon and follow the on-screen installation instructions. You will need administrative privileges to install the software.

Once the installation is complete, close the installer, open the Mac OS X /Applications folder, and double-click on the XDarwin icon. The first thing you will see is a screen asking which display mode you wish to use: Full Screen (Rooted) or Rootless (see figure 2.16).

If you choose Full Screen, XDarwin takes over the entire screen; if you choose Rootless, Aqua and XDarwin coexist on the same screen. To switch between the full screen mode and Aqua, press Command+Option+A. To exit XDarwin, locate the main `xterm` and type `exit`, or switch to Aqua and select Quit from the XDarwin menu.

Now that the X server is running, you have full access to UNIX X11 applications. You can run local X11 applications or `ssh` to a remote host and run them from there, just as you would in a traditional X session. XDarwin's default window manager is `twm`, but you can install and use others (including the old favorite, `fvwm`).

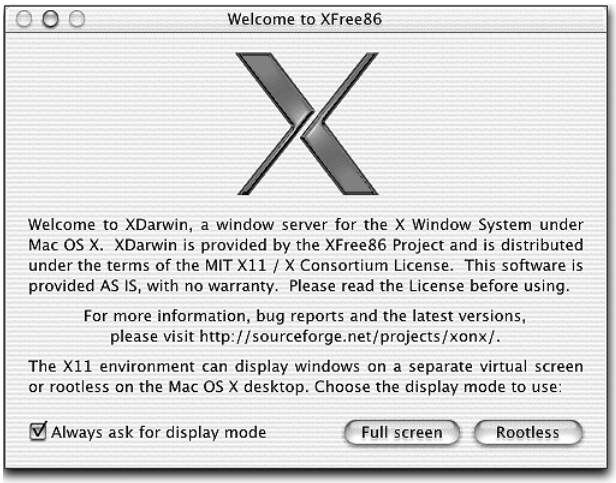

Figure 2.16
If you run XDarwin in Full Screen mode, XDarwin takes over the entire screen. In Rootless mode, X Window and Aqua coexist on the display simultaneously.

2.16 UNIX to Mac OS X software projects

Mac OS X users are fortunate to have access to a large amount of UNIX software that can run under Mac OS X systems. In addition, many projects are devoted to bringing UNIX tools to the Mac OS X environment. Mac OS X ships with many commonly used UNIX commands and tools, including `ls`, `cat`, `more`, `emacs`, `vi`, `top`, and Perl. Installing the development tools (discussed in appendix A) adds more tools to the system, including `gcc`, `g++`, `gdb`, `RCS`, and `CVS`.

To install tools not covered by these methods, you have several choices. Mac OS X is based on BSD, after all, so in most cases you can simply locate and download a tool's source distribution and compile and install the software in the usual manner. Several projects attempt to simplify this process by providing software tools and infrastructure that help you locate, build, and install UNIX tools on the Macintosh:

- *The Fink project* (http://fink.sourceforge.net)—Provides UNIX tools to Mac OS X users by porting existing UNIX tools to the Mac OS X environment and then making these ports available to the public through packages. You first download the client-side package manager software (called fink) and install it on your system. The fink software presents a list of available software packages. You select packages from the list, and fink handles all the installation details, including downloading the package and package dependencies, building the software, and installing it on your system. This process

simplifies many of the tricky issues associated with porting and installing software packages. Another nice feature is that fink places all installed software into a separate directory, away from the system files—you never need to worry about clobbering system files when installing new software.

- *The GNU Mac OS X Public Archive* (http://www.osxgnu.org)—(Specifically, the OSXGNU Software Archive.) Contains ports of many UNIX tools in Mac OS X package format (the standard format used to install software under Mac OS X). To install a tool, you locate it on the site, download it, double-click on the tool package icon, and follow the on-screen instructions.
- *The Darwin Collection* (http://www.ptf.com/tdc)—A set of CDs that contains the Darwin OS and Mac OS X ports of the BSD Ports Collection and GNU software. The most interesting aspect of this collection is the Mac OS X port of the BSD Ports Collection. Currently, the BSD Ports Collection contains nearly 4,000 packages. Each package contains the most up to date, stable version (source code, patches, and so on) of the software. Users can install individual packages or the entire package tree from either source code or from premade binaries. The Mac OS X ports bring these tools and services to Mac OS X.
- *The Apple developer site* (http://developer.apple.com/unix/index.html)—Contains a wealth of information about UNIX development and tools for Mac OS X.
- *Open Darwin, Darwin Ports* (http://www.opendarwin.org/projects/darwinports)—Provides many software ports for the Darwin system.

In addition to these projects, many others are committed to bringing UNIX tools to Mac OS X.

2.17 Summary

In this chapter, you encountered some of Mac OS X's UNIX tools and saw how to accomplish familiar UNIX tasks under Mac OS X. Armed with this knowledge, you are ready to learn about Mac OS X development tools such as Project Builder and Interface Builder, which enable you to easily develop Mac OS X command-line and GUI applications.

Part 2

Tools

Chapters 3 and 4 introduce you to programming Mac OS X. The chapters provide information on Apple's freely available development tools and its main development environment. Throughout this section, you will learn what tools are available and how to use them in your own projects. What's more, you will be able to see many of the tools in action through practical examples.

3 Project Builder and Interface Builder

- Macintosh IDEs
- Using Project/Interface Builder
- Setting up CVS for Project Builder
- Configuring optimization for Project Builder
- Static code checking for Project Builder

All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer.

—IBM maintenance manual (1925)

The Macintosh has always supported application development with some very good development tools, Project Builder and Interface Builder among them. This chapter provides an overview of the Project Builder environment, its use, and covers common scenarios you will encounter when developing programs with Project Builder. Once you complete this chapter, you will be able to work the levers of Project/Interface Builder and will be able to get around in the environment.

3.1 Introduction

Traditionally, Macintosh development tools are centered on an *Integrated Development Environment* (IDE). An IDE is commonly composed of an integrated editor, compiler, linker, and debugger, all within one program. To develop a new application, you launch the IDE and create a new project file. The project file acts as a repository for all files that make up the project, including source files, libraries, and any support files. You write your program using the integrated editor, build the program by selecting a build command, and execute the program with a run command. Typically commands are accessible from menu items, and customization takes place through standard dialog boxes. Debugging a program is as simple as building the program in debugging mode and stepping through the program within the IDE. When encountering an error, you simply edit the code (in place), rebuild, and continue debugging the program.

The strength of this approach is that all tools and commands are accessible through a consistent user interface. Also, you can easily access hard-to-remember commands and options from menus and dialog boxes.

UNIX, on the other hand, has always offered a more segregated development environment. To create a new project, you first create a makefile, specifying what files compose the project, as well as the build tools, their options, and any numbers of build commands. You write the program using your favorite editor, and build and run the program from a shell. To debug the program, you run it within a command-line debugger (`gdb`), run it within a debugger in `emacs`, or use print statements.

3.1.1 Macintosh Programmer's Workbench

One of the first development environments for the Macintosh was Macintosh Programmer's Workbench (MPW), from Apple. MPW is an advanced environment for developing applications for 68k and PowerPC machines running Mac OS. It contains all the tools you would expect from an advanced development environment, including an editor, compiler, build tools, debugger, and shell. MPW was separated from other Macintosh development environments of its time by the blending of a command-line environment with elements of an IDE. To perform tasks and development activities, you entered commands into *worksheets*, much as you would under a UNIX shell, although the command language was specific to MPW. Figure 3.1 shows an MPW worksheet with some basic commands.

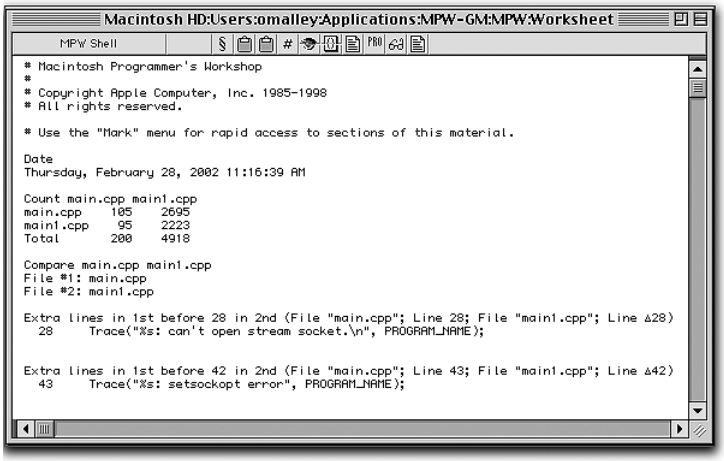

Figure 3.1 MPW provides a command-line interface for entering commands and executing development tasks.

MPW stood somewhere between a pure command-line development environment and an IDE. It required a bit of a learning curve compared to other IDEs, but it was far more powerful and extendable; it also was a favorite among advanced developers and those who preferred a command-line interface for application development. If you are interested, MPW is freely available from Apple at http://developer.apple.com/tools/mpw-tools.

3.1.2 THINK Pascal and THINK C

Another popular series of development tools for the Macintosh included the THINK Pascal and THINK C development environments from Symantec. Both

THINK Pascal and C were very good IDE-based development tools used by most Macintosh developers. The THINK Pascal debugger was one of the best parts of the program, foreshadowing many of the features that appeared in later Macintosh debuggers. In addition to their development tools and productive user interface, both environments supplied all the necessary software infrastructure for building Macintosh 68k applications. Later incarnations of THINK C included a C++ compiler and application framework called Think Class Library (TCL).[1]

You can still get a free copy of THINK Pascal from ftp://ftp.symantec.com//public/english_us_canada/products. Like MPW, it runs under Classic mode and is mainly of interest for its historical value or support of legacy applications.

3.1.3 CodeWarrior

Throughout the late 1980s to mid-1990s, the THINK tools were very popular. About this time, a company called Metrowerks began producing development tools for the Macintosh under the name CodeWarrior. The CodeWarrior environment was similar to the THINK tools; it included an editor, compilers, debugger, as well as an application framework called PowerPlant. At that time, the main features that distinguished CodeWarrior from other environments were its productive user interface, the quality of its compilers, and how it supported different compilers within a single development environment. In addition, Metrowerks was first to release a PowerPC (PPC) compiler when Apple transitioned its product line from the 68k to the PPC architecture.

THINK Pascal and THINK C were two separate products with two separate, yet similar, interfaces. In contrast, CodeWarrior offered a single development environment that supported different compilers, all within one product. Over time, Symantec lost market share to Metrowerks as the development environment of choice for building Macintosh applications. Metrowerks' CodeWarrior is alive and well and is still one of the best commercial development tools for developing Macintosh programs (http://www.metrowerks.com).

3.1.4 Project Builder and Interface Builder

During this time, several attempts were made to bring UNIX tools to the Macintosh, although they were never mainstream efforts whose goal was to compete

[1] THINK C was never a true C++ compiler. It lacked many C++ features such as constructors and method and operator overloading. Symantec C++ was the first true C++ compiler from THINK/Symantec, but it was too little too late, because most developers had already switched to Metrowerks compilers.

with commercial products or offer the broad feature set of MPW, THINK Pascal and C, and CodeWarrior. Before Mac OS X, you could find usable implementations of UNIX tools including Perl, `gcc`, `bison`, `flex`, and `sed` for the Macintosh. However, many of these tools never integrated well with the platform. With the introduction of Mac OS X, you now have access to a broad range of UNIX-based development tools from Apple, as well as third-party and open source developers. These tools do integrate well with the Mac OS X environment and provide a solid development foundation for building applications under Mac OS X.

Apple provides two main development tools for building applications under Mac OS X: Project Builder and Interface Builder.

Project Builder

Apple's Project Builder is a freely available IDE that contains an editing, build, and run environment for developing Mac OS X applications. With Project Builder, you can build all types of Mac OS X applications, including Carbon and Cocoa applications, bundles, frameworks, kernel extensions, Java applications and applets, plug-ins, and tools (don't worry if you do not know what some of these terms mean; they are all covered later in the chapter).

Project Builder is in the tradition of an IDE in the sense that all development tools and commands are aggregated under a single program. However, it does not include the main development tools (compiler, linker, assembler, version control, etc.) as part of the program. Instead, it uses UNIX development tools such as `gcc`, `g++`, and `gdb`. In a sense, Project Builder is evolutionary: it continues the line of IDE-based development environments for the Macintosh, but breaks with tradition by using external UNIX-based build tools for implementing build and development tasks. It strikes a nice balance by providing a modern interface for application development while leveraging the strengths of the UNIX tools set.

Interface Builder

Apple's Interface Builder is used to design the user interface component of your program. Using Interface Builder, you design your application's user interface components including menus, windows, icons, and dialog boxes. In addition, you can use Interface Builder to create your program's code framework, which you fill in using Project Builder.

3.2 Creating an application with Project Builder

Before jumping into the details of Project Builder, let's begin by looking at how simple it is to create an application. As you will see, Project Builder enables you to get a basic application shell running in no time. The example application you will build is a Cocoa program that displays an image in a window. You will learn all about Cocoa in chapters 5 and 6, but for now think of it as a collection of object-oriented libraries, or *frameworks*, for constructing GUI- and non GUI-based applications for Mac OS X.

Throughout the book, you will develop many applications. For consistency, you'll store all the projects under a directory called projects, located in your home directory. At this point, create a new folder in your home directory and name it projects. Now, follow these steps:

1. Move to the Developer/Applications folder and launch Project Builder. (Better yet, drag the Project Builder icon to the Dock so you can get at it easily.)
2. Choose File→New Project (Shift-Command-N), select Cocoa Application (Nib Based)[2] from the New Project list (see figure 3.2), and click the Next button.

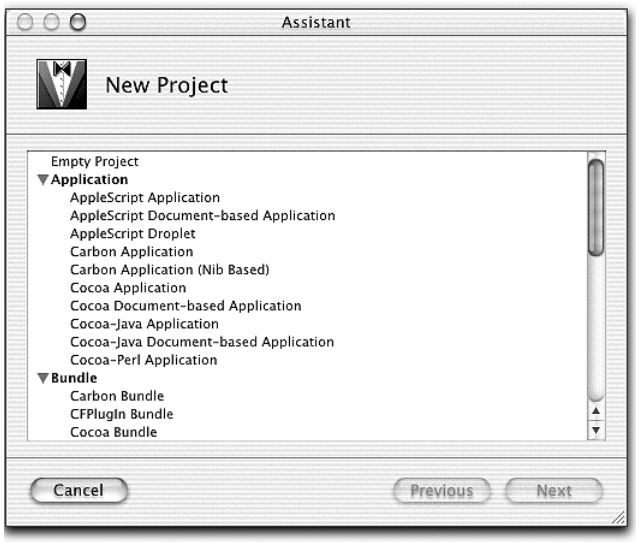

Figure 3.2
The New Project list displays all project types you can build within Project Builder.

[2] For Cocoa-based applications, a Nib file holds your application's interface objects (windows, menus, and so on) as well as the objects' attributes and runtime relationship to other objects.

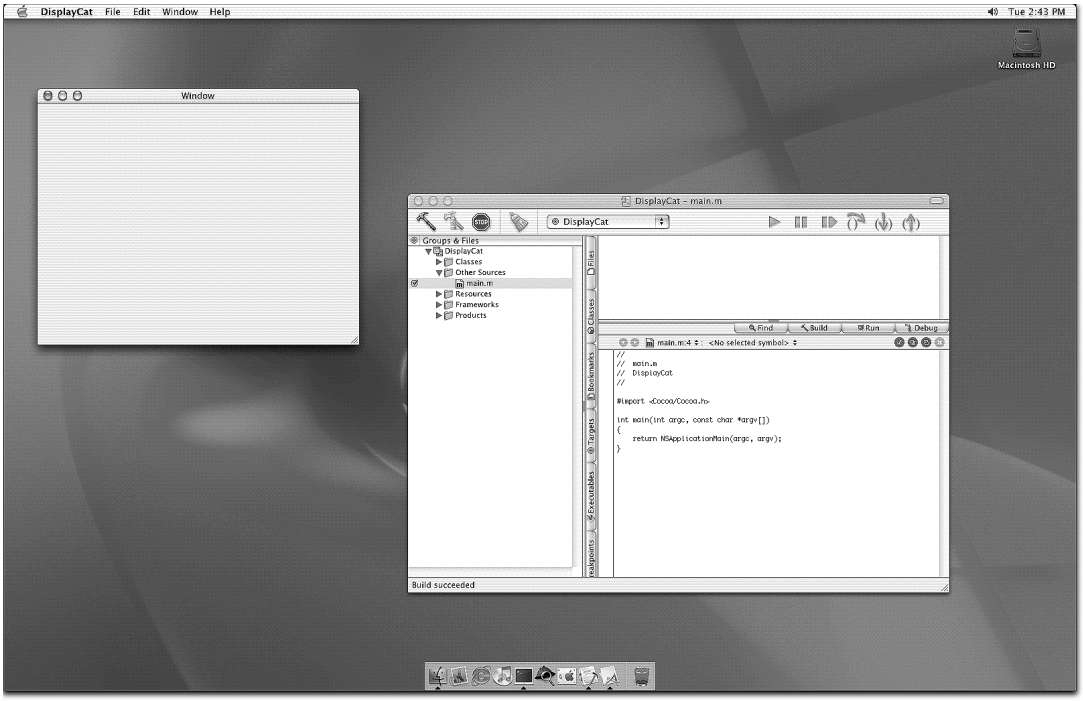

Figure 3.3 The DisplayCat program before you add anything to the project

3. Set the location to your projects folder and the project name to DisplayCat, and click the Finish button. Project Builder will create a new project and display its main window.

Before making any changes, build and run the program by selecting Build→Build and Run (Command-R). As you can see in figure 3.3, with no coding you have a working application complete with a window and menu—all for free. Press Command-Q to quit the program.

Next, let's add a picture to the project:

1. Select the Resource group, located in the Contents pane (on the left side of the main window under Groups & Files), and select Project→New Group. Call the new group Images (see figure 3.4).
2. Drag the file cat.tiff from DisplayCat/Images (located on the source code distribution disk) to the DisplayCat folder within your project directory.

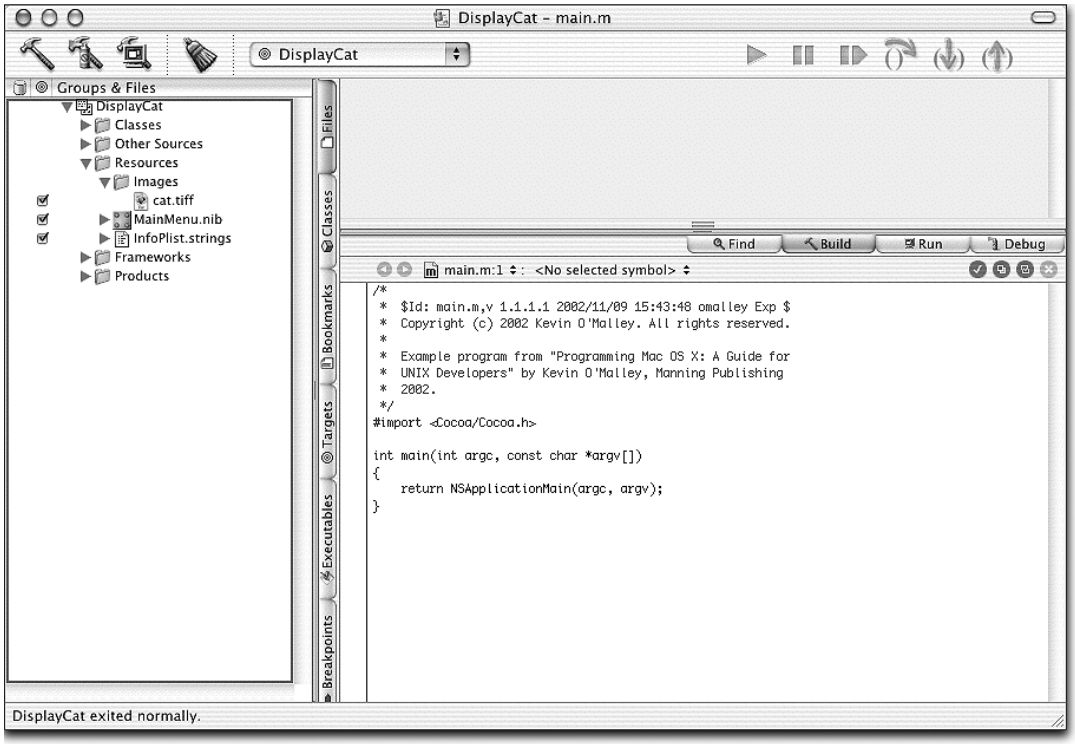

Figure 3.4 The Contents pane holds project file, libraries, and resources such as images. The Images group contains the cat.tiff file that the program displays on the main window.

3. Select the Images group and choose Project→Add Files. Select the cat.tiff file and click the Open button. Click the Add button in the next dialog to add the image file to the project. Doing so adds a picture of a cat to the project so it can be displayed on the program main window.

The next step is to add the cat picture to the main application window of the project:

1. If necessary, expand the Resource group (in the Contents pane) and double-click on the MainMenu.nib file. Doing so launches Interface Builder and loads the program's MainMenu.nib file. You'll use Interface Builder to design your application's user interface, including menus, windows, icons, and dialog boxes. You will see four open windows within Interface Builder, as shown in figure 3.5. The first window (titled Window) is where you place the application's picture. To its right is the Palette window, which holds Application Kit interface components. The window at the

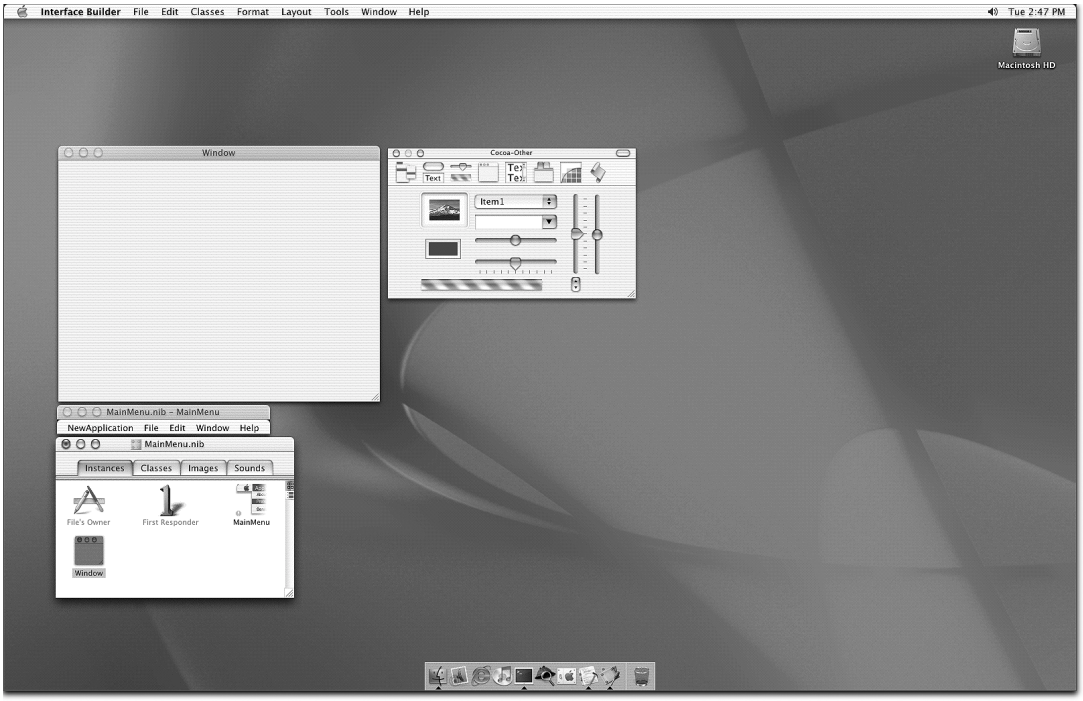

Figure 3.5 The four Interface Builder windows enable you to construct your program's user interface.

bottom of the screen, called MainMenu.nib, holds the definition for the application menu, class instances, and images and sounds for the application. It also contains more complex information that is described in detail in chapters 6 and 7. Above this window is the application's main menu.

2. Click on the Cocoa-Other icon, located in the toolbar of the palette window (third icon from the left), and select and drag the NSImageView object to the main window.
3. Move the NSImageView object toward the upper left in the window until you see the Aqua guides. The Aqua guides become active when you drag an interface object within a window. They provide on-screen feedback for placing an interface element in the correct location as specified in Apple's Aqua Human Interface Guidelines. By following the Aqua guides, you can be sure you place interface elements such as buttons and text fields correctly within the window, adhering to Apple's interface guidelines.
4. Using the Aqua guides for placement, resize the object until it fills the entire window (see figure 3.6).

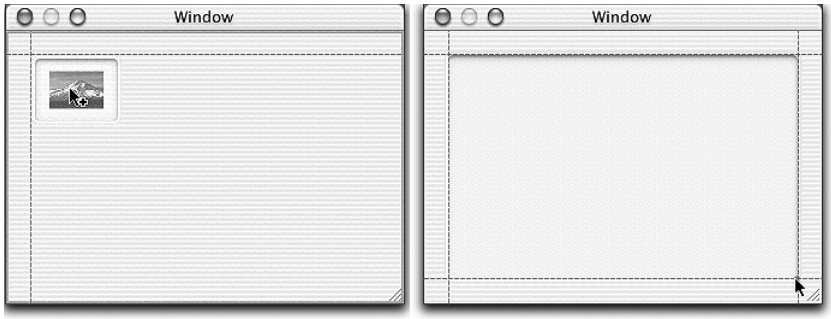

Figure 3.6 To add the picture, drag an NSImageView object from the palette window and resize it to fill the entire window.

5 Click the Image tab on the MainMenu.nib window and drag the image of the cat to the NSImageView object on the main window (see figure 3.7).

6 Save your work and switch back to Project Builder.

7 Build and run the project.

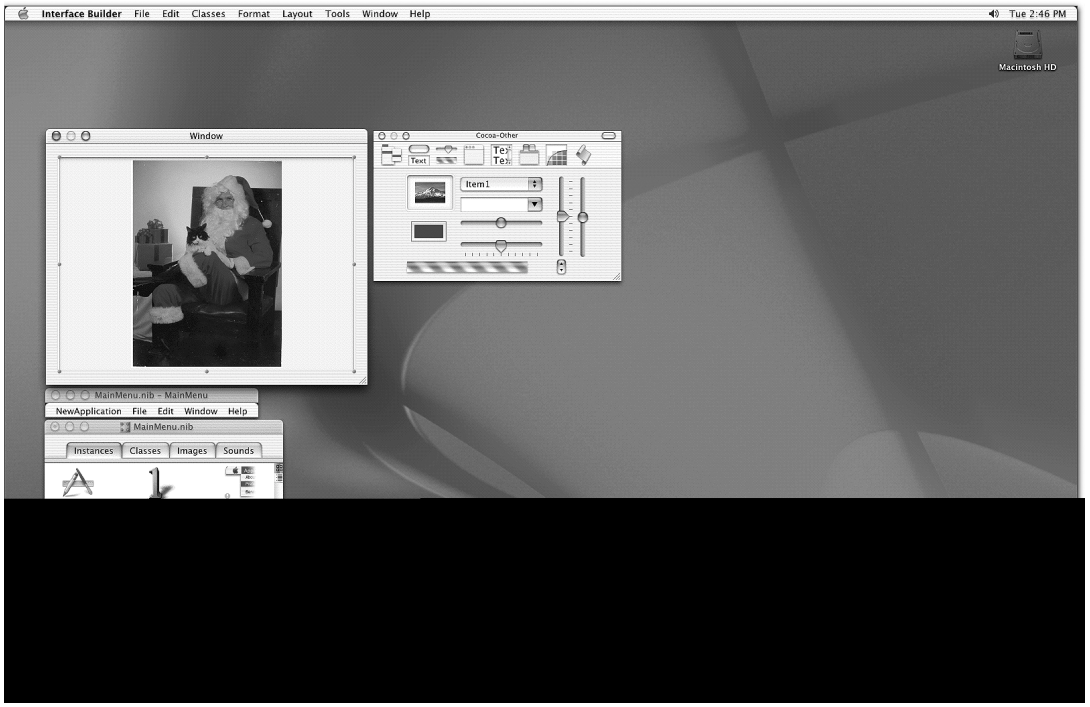

Figure 3.7 The main window after adding the picture to the NSImageView object

There you have it. In a few simple steps, you have a fully functioning application, complete with an application menu and window. As this example demonstrates, Project Builder gives you all the tools and infrastructure you need to construct applications with little effort. In fact, for this example you did not even write a line of code!

3.3 Project Builder in depth

As you saw in the previous section, creating the core infrastructure for an application is simple and straightforward with Project Builder. With just a few clicks, you were able to get a basic application running in no time. Next, let's look at Project Builder in detail, discussing its features and operation.

3.3.1 Targets and build styles

Before describing Project Builder's interface, let's define some terms and concepts. First, you need to understand targets and build styles. A *target* collects project components that make up a project (source files, header files, and libraries), defines the basic instructions and attributes that specify how Project Builder builds a project, and holds information a program uses at runtime, such as command-line arguments and environment variables. Think of a target as a way of encapsulating the items and attributes that form a program. *Build styles*, on the other hand, override certain aspects of a target's build instructions to create specialized versions of the program.

Let's illustrate targets and build styles through an example. Imagine you wish to develop a program called `AgentServer`. The program reads XML-formatted messages from agents, performs some action, and returns a result to the agent. One of the primary requirements of the program is performance: it must service agent request in a timely and predictable manner. To test this requirement, you write several test agents that simulate various agent behaviors and create different versions of your program; each build has different compiler optimizations. The goal is to test these agents with each version of the program and see how they perform.

Let's take a high-level look at how you can use targets and build styles for this problem:

1. Create a new project for the program. When you create the new project, Project Builder also creates a new default target with the same name as the project. This target automatically holds all files that compose the program, as well as default build settings.

2. Create several new build styles—one for each type of optimization setting you wish to test.
3. To get performance statistics, you also need to add profiling code, either by writing your own routines or by using the `-pg` compiler flag. The `-pg` flag adds profiling code that produces an execution profile of your program. GNU `gprof`, a profiling tool that comes with Mac OS X, uses this information to display performance statistics for your program. (This program, as well as the other Mac OS X developer tools, is discussed in chapter 4.)

Testing the different versions of the server is as easy as selecting a build style, building the program, running the program and the test agents, and collecting statistics. You repeat this process for each build style. Once you've finished, you can compare the effect of the different optimization settings on the program's performance. As you can see from this example, build styles are a simple and intuitive way to apply different build settings to a target.

Let's take this example a step further. Imagine your server uses a third-party XML parser to decode the XML-based strings sent by the agents. Also imagine you've wrapped the parser with custom code that encapsulates its behavior, so swapping in a different parser will not change any client code. After repeated testing, you find the parser is the performance bottleneck. At this point, you would like to swap in some other parsers (maybe one you wrote) to see if you can increase performance. This is a perfect application for using multiple targets within a project. Without targets, you would need to create several new projects, one for each parser. Using targets, you create a new target for each parser you wish to test, add the appropriate files to the target, and switch between each target for testing—all within one project.

To sum up, targets collect files and build settings for a program. If different versions of your program contain different files, you should express each version with a target. Build styles enable you to override the default build setting for the active target so you can perform particular types of builds.

3.3.2 Project Builder's UNIX tools

Another concept to understand is that Project Builder uses UNIX command-line tools for performing builds, managing source code, and debugging applications. Thus you can leverage your current knowledge of UNIX development tools when using Project Builder. For example, Project Builder contains all the hooks for adding stricter static checking to your compiles. So, you can use the same compiler flags you use from the command line within Project Builder.

3.3.3 *Project Builder's interface*

Now, let's take a closer look at each of the components of the main project window. Figure 3.8 shows the main Project Builder window.

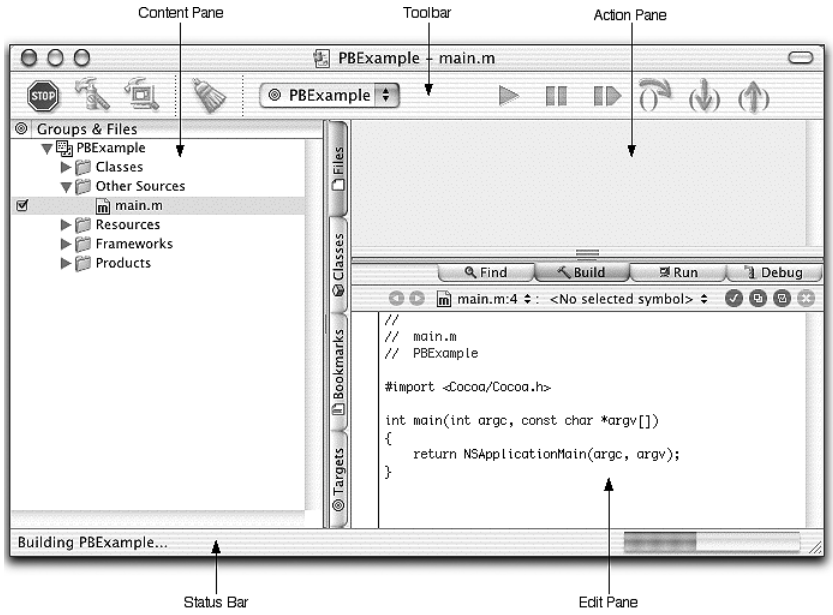

Figure 3.8 The main Project Builder window

The toolbar

At the top of the window is the toolbar, which contains a series of icons representing common Project Builder commands (see figure 3.9).

The icons on the left side of the toolbar execute various build commands. Beginning at far left, the buttons are as follows:

- *Build Active Target*—(Command-B) Builds the active target by executing the `build` command.
- *Build and Debug Active Target*—(Command-Y) Executes a `build all` command and runs the program under the debugger.
- *Build and Run Active Target*—(Command-R) Similar to Build Active Target; but once Project Builder successfully builds the project, this button runs the program.

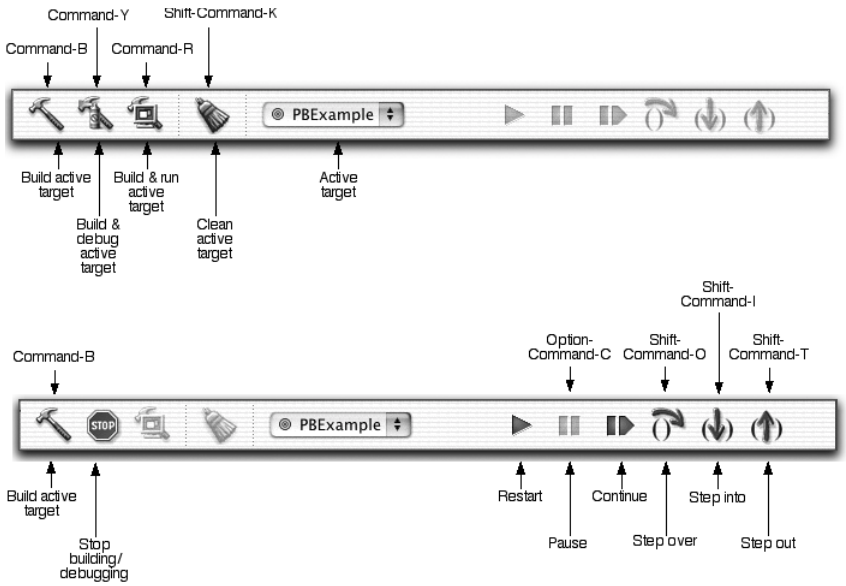

Figure 3.9 The toolbar contains shortcuts to common build and run commands.

- *Clean Active Target*—(Shift-Command-K) Invokes the `clean` command, deleting any intermediate files from the project folder (including object and executable files). This command is like the UNIX `make clean` command.

Collectively, these icons enable easy access to the most frequently used build commands.

The Active Target menu enables you to move between all active targets in your project. A target encapsulates the items that compose a version of the program and the general attributes that define how Project Builder builds these components.

The buttons on the right end of the toolbar run debugging commands. Project Builder enables these icons once a program is running under the debugger. Beginning from the left, they're as follows:

- *Restart*—Restarts, or reloads, the program in the debugger (does not continue from where you were, but from the beginning of the program).
- *Pause*—(Option-Command-C) Suspends execution of the current program.
- *Continue*—Resumes program execution.

The next three commands control how execution continues from a function or method call:

- *Step Over*—(Shift-Command-O) Executes the current function or method, but does not single-step into the function code.
- *Step Into*—(Shift-Command-I) Jumps to the current function or method and single-steps through the code.
- *Step Out*—(Shift-Command-T) Immediately returns to the calling function after executing the rest of the function.

The Contents pane

The next component of the project window is the Contents Pane, shown in figure 3.10. The Contents pane provides various views of project items, as well as easy access to the individual items that compose the target:

Figure 3.10
The Contents pane provides access to project file, libraries, and resources such as images.

- *Files view*—Accessible by clicking on the Files tab. Lists all files and components that make up the project. These include source and header files, libraries, frameworks for the application environment (Carbon, Cocoa, Java), resource files, and the project product, or executable application. Where applicable, clicking or double-clicking on a file will open it for editing or viewing. For example, clicking on a source file or a header file will display it in the Editor pane. Double-clicking on a .nib file will open the file in Interface Builder.
- *Classes view*—Enables easy access and browsing of application and framework class files (see figure 3.11). The Class pane (upper pane) displays a filtered

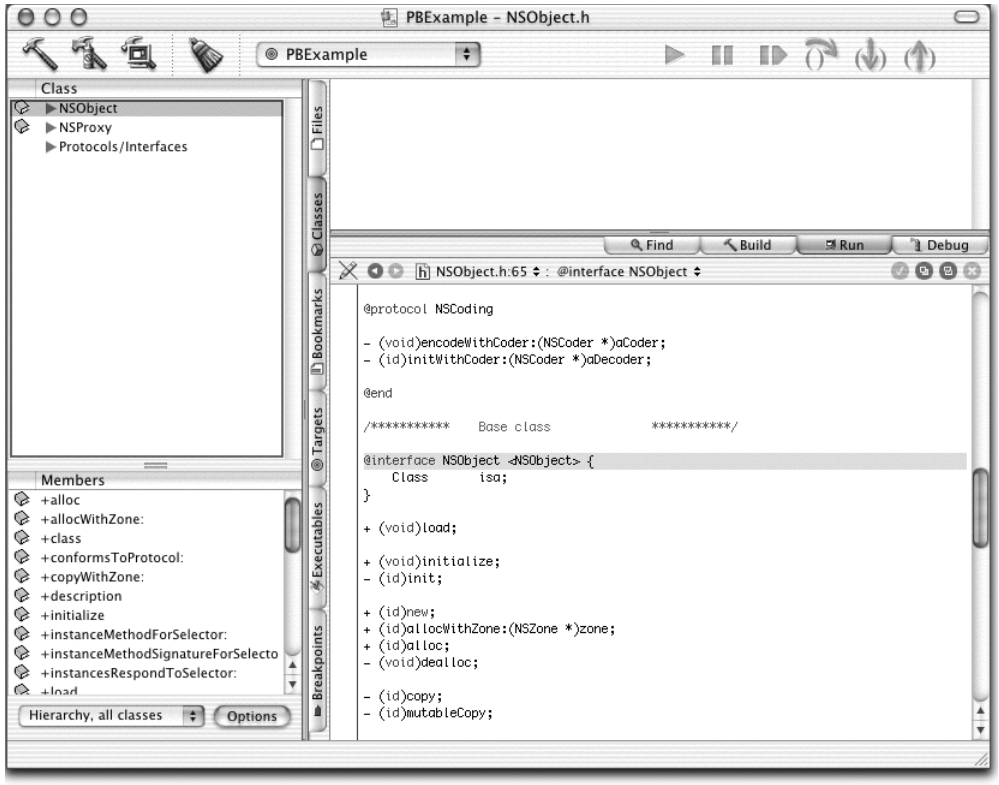

Figure 3.11 The Classes view lets you browse the interface and implementation class files for both application and framework classes.

view of all classes in the active target. You can filter the files that Project Builder displays by selecting various filtering options from the pop-up menu at the bottom of the pane. You can also change the display options by clicking the Options button, also located at the bottom of the pane. (You have not created any new classes in the `DisplayCat` project, so certain filtration options may not show anything.) Double-clicking on a class's book icon displays documentation for the class. Clicking on a class displays its members in the Members pane (lower pane). The Members pane lists all members for the selected class. Clicking on a class member loads the class's implementation file into the editor. Double-clicking on a class member loads the class's interface file into the editor in an external window.

Figure 3.16 The Current View menu lets you select an element to display in the Editor pane.

- The *Go Back* and *Go Forward* icons at the left end of the toolbar enable you to cycle forward and backward between currently loaded views.
- The *Current View* item displays the currently loaded entry. Clicking on this item displays a menu of all loaded views (see figure 3.16). From here, you can select different views to display in the pane.
- The *Current Location* item shows the cursor position within the current file. For example, if a source file is loaded and the cursor is on or within a method, Project Builder displays the method name. Like Current View, clicking on this item will show a menu that lists the file's functions or methods (see figure 3.17).
- Project Builder enables the *Check Syntax* icon when a source file is loaded. Clicking on this button will check the syntax of the file based on its language.
- The *Display New Counterpart Syntax* icon toggles between interface and implementation files. If an implementation file is loaded, clicking this button displays its interface file; if an interface file is loaded, clicking this button displays its implementation file.

Figure 3.17 You can move to a different function or method in a source file by choosing its name from the function menu.

- The *Split Editor* button splits the Editor view into panes. Clicking on the *Close Split* icon closes the current pane.

The status bar

The status bar is located at the bottom of the project window. It displays information about the status of Project Builder operations, including the stages and results of a project build, the result of a find operation, and other tasks.

3.3.4 Project Builder scenarios

Now that you understand some of the basics of the Project Builder interface, let's take a closer look at several scenarios that consistently come up when developing programs under Project Builder. Remember, programming, like most aspects of computing, is learned through practice, not just by reading or studying theory. Theory may be able to get you from linear to logarithmic, or exponential to linear time, but it cannot teach you to write good code—you accomplish this through practice. Consequently, be sure you try these examples as you read. Enough talk; let's get down to work.

Creating a project

The first step in developing a program under Project Builder is to create a project. Project Builder enables you to create many kinds of programs for Mac OS X, including applications written in Carbon, Cocoa, Java, frameworks, bundles, and kernel extensions. Creating a project is similar to writing a makefile, but provides a friendlier way of controlling how a project is built and what is included in the build.

To create a new project, launch Project Builder and select File→New Project (Shift-Command-N). Project Builder opens a window that displays a list of the available project types (see figure 3.18).

Let's take a closer look at the different project types. Currently, you can choose from seven categories plus the Empty Project option:

- *Empty Project*—A project with no added libraries, frameworks, or other software infrastructure files.
- *Application*—There are nine application types:
  - *AppleScript Application*—An AppleScript Studio application, which is a Cocoa-based application that contains hooks for AppleScript. This type of project is useful for putting a Cocoa-based GUI on an AppleScript-based program.
  - *AppleScript Document-based Application*—The same as an AppleScript Application, but includes support for the NDDocument architecture.

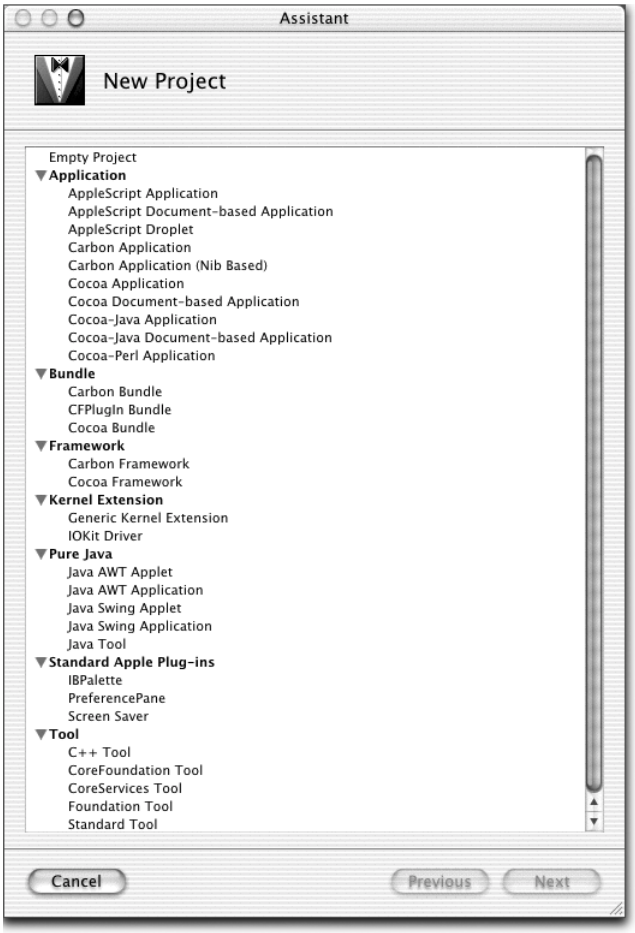

Figure 3.18
The New Project assistant list displays all project types you can build within Project Builder.

- *AppleScript Droplet*—An AppleScript application in which files are processed by dragging them to the application icon.
- *Carbon Application*—A Carbon application that includes all necessary support files and frameworks for developing both Mac OS and Mac OS X Carbon applications; it uses Resource Manager files to store application resources (.r or .rsrc files).
- *Carbon Application (Nib Based)*—The same as Carbon Application, but uses Nib-based resources (.nib).
- *Cocoa Application*—A Cocoa application using Objective-C as its development language.

  - *Cocoa Document-based Application*—Same as Cocoa Application, but adds support for the NDDocument architecture.
  - *Cocoa-Java Application*—A Cocoa application using Java as its development language.
  - *Cocoa-Java Document-based Application*—Same as Cocoa-Java Application, but adds support for the NDDocument architecture.
- *Bundle*—There are three bundle options:
  - *Carbon Bundle*—A bundle linked to Carbon.
  - *CFPlugin Bundle*—A bundle linked to the Core Foundation framework.
  - *Cocoa Bundle*—A bundle linked to Cocoa.
- *Framework*—There are two framework types:
  - *Carbon Framework*—A framework linked to Carbon.
  - *Cocoa Framework*—A framework linked to Cocoa.
- *Kernel Extension*—There are two kernel extension options:
  - *Generic Kernel Extension*—A kernel extension project.
  - *IOKit Driver*—An I/O Kit project, used for developing kernel drivers.
- *Pure Java*—There are five Pure Java types to choose from:
  - *Java AWT Applet*—A project for developing AWT-based (Abstract Window Toolkit) Java applets. AWT is superceded by Swing, which is more advanced and simpler to use. It is used primarily for compatibility with Mac OS 9.
  - *Java AWT Application*—A project for developing AWT-based (Abstract Window Toolkit) Java applications. AWT is superceded by Swing, which is more advanced and simpler to use. It is used primarily for compatibility with Mac OS 9.
  - *Java Swing Applet*—A project for developing Swing-based Java applets. Swing supercedes AWT and provides a more advanced and simpler to use interface toolkit.
  - *Java Swing Application*—A project for developing Swing-based Java applications. Swing supercedes AWT and provides a more advanced and simpler to use interface toolkit.
  - *Java Tool*—A project for developing Java applications or libraries that do not require a GUI.

- *Standard Apple Plug-ins*—There are three options for standard Apple plug-ins:
  - *IBPalette*—A project for developing an Interface Builder palette, which contains components available for developers to add to applications (including menus, text fields, and other interface components).
  - *PreferencePane*—A project for developing a Preference Pane, which resides in the System Preference application and is used to set system-wide parameters such as screen saver settings, network settings, and energy settings.
  - *Screen Saver*—A project for developing screen saver modules.
- *Tool*—There are five command-line tool types:
  - *C++ Tool*—A project for developing C++ applications. This option is used for building C++ command-line tools.
  - *CoreFoundation Tool*—A project for developing a tool that is linked to the Core Foundation framework.
  - *CoreServices Tool*—A project for developing a tool that is linked to the Core Services framework.
  - *Foundation Tool*—A project for developing a tool that is linked to the Foundation framework.
  - *Standard Tool*—A project for developing C applications. This option is used for building C command-line tools.

To create a new project, select the appropriate project type from the list and click the Next button. Enter the name of the project and the location where the project folder will be stored. You can select a location by clicking the Choose button and choosing the location from the directory sheet. Once you have filled in this information, click the Finish button. Project Builder will create a new project based on a template and store it in the specified location.

Adding files to a project

A common operation during program development is to add files to a project. Project Builder supports this process through the New File command.

Imagine you have already created a new Cocoa project, written in Objective-C, and you wish to add a new source file to the project. Follow these steps:

1. Select File→New File or press Command-N to open the New File window shown in figure 3.19.

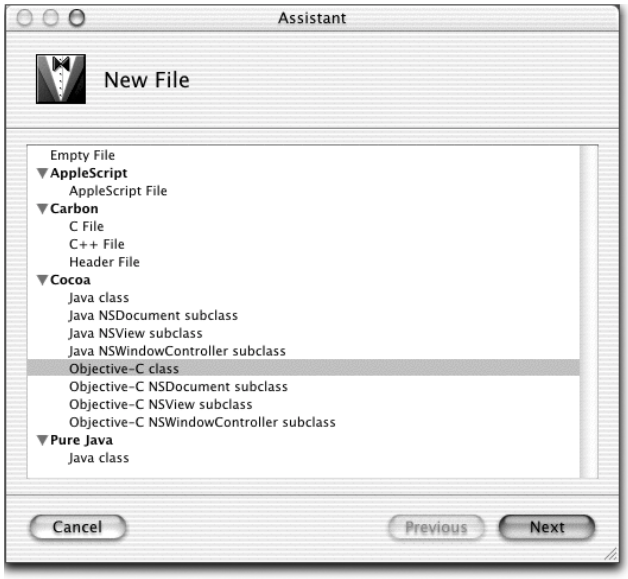

Figure 3.19
The New File window displays the types of files you can create for various languages and projects.

2 The New File window lists the types of files you can add to your project, classified by project category. Let's say you wish to add a new Objective-C class to a Cocoa project. To do this, select Objective-C Class from the list and click the Next button to open the window shown in figure 3.20.

Figure 3.20
When creating a new class file, Project Builder displays the New Objective-C class window. Use this window to specify the name of the class and its location.

3. Enter the name of the class. Make sure the Also Create checkbox is checked so a corresponding header file is created.
4. Select the location where the generated class files will be stored (typically the project folder). You can select a location by clicking the Choose button and choosing the location from the directory sheet, or you can type it in by hand. The Add To Project menu lets you select the project to which the new files are added (typically the current project).
5. Click the Finish button. Project Builder creates new files for the class's interface and implementation, stores them in the specified location, and adds them to your project.

The new files are accessible from the Contents pane. By default, the files are located outside the listed folders. Usually, you will move the new class files to the Classes folder by highlighting them and dragging them to the folder.

Using CVS

As you already know, Project Builder uses UNIX tools to perform many of its tasks. For version control, Project Builder uses CVS (Concurrent Versions System). The CVS revision control program stores a file's change history and supports commands for easy access to past versions of the file. CVS is built on top of a version control system called Revision Control System (RCS), and uses RCS commands behind the scene to perform its actions. (The RCS program dates back to the early 1980s and was written by Walter F. Tichy while at Purdue University.) Though the underpinnings of CVS and RCS are similar, the nomenclature, intended audience, and command set are very different.

Both RCS and CVS are excellent choices for a version control system and are available under Mac OS X. The system you use really depends on the organization of your project. Project Builder supports CVS from its interface. Unfortunately, it does not currently support RCS.

Setting up CVS for use with Project Builder is simple: you set up a CVS repository on one of your disks, set your CVS environment variables, and check in/out the project. Once these steps are complete, you can open the project under Project Builder and get full access to the CVS command set and repository. To make this clear, let's go through each step in more detail.

Before using CVS, you need to configure a few things, including the CVS repository and the client environment. The first step is to set up the CVS repository and environment variables:

1. Open the Terminal application (located in /Applications/Utilities) and create a directory to hold the CVS repository for your projects. The repository is a central location that holds all files stored under version control. Place this directory on a disk partition that is accessible to all users of the version control system and that is large enough to handle the anticipated file storage requirements. Try to be overly conservative when estimating your disk requirement. For this example, place the repository in your home directory under the name cvs-repository.
2. Set the CVS environment variable `CVSROOT` to the location of the directory holding the repository (the directory you just created). Doing so enables CVS commands to locate files under version control. The following command sets the `CVSROOT` environment variable to the correct location (for the `tcsh` shell).

   ```
   % setenv CVSROOT /Users/omalley/cvs-repository
   ```

3. The `CVSEDITOR` environment variable specifies the editor you will use to enter file revision descriptions. Set it to your favorite editor, such as `emacs` or `vi`. Under Project Builder, this variable is not used; instead, Project Builder opens a Sheet dialog in which you enter revision commands. If you will ever use CVS from the Terminal, it makes sense to set the variable anyway. Because I'm an `emacs` user, I set this variable to `emacs`:

   ```
   % setenv CVSEDITOR emacs
   ```

4. Run the CVS initialization command to create the CVS administrative files in the repository:

   ```
   % cvs init
   ```

You only need to run the `cvs init` command once, before anyone on the system uses the new repository. If for some reason you set up another repository in a different location, say for personal files, just change the `CVSROOT` environment variable to the location of the new repository and run the CVS initialization command.

For convenience, add the environment variables to your startup file. Doing so prevents you from entering them each time you open a shell. For the `tcsh` shell, add them to your .cshrc file:[3]

[3] I typically do not set my CVSROOT in .cshrc. Instead, I set the repository location manually using aliases:

```
alias cvs-proj1 'setenv CVSROOT /Users/omalley/proj1'
alias cvs-proj2 'setenv CVSROOT /Users/omalley/proj2'
```

This approach enables me to switch between multiple project repositories.

```
% setenv CVSROOT /Users/omalley/cvs-repository
% setenv CVSEDITOR emacs
```

Now, the environment is set up and ready to go. The next step is to place a project under version control and access it from Project Builder. You use the `cvs import` command the first time you place a module under version control. The import command takes all files in the working directory (and subdirectories) and adds them to the repository specified by `CVSROOT`:

```
cvs import [-options] [repository] [vendortag] [releasetag]
```

The `options` argument specifies options applied by the import command. (See the CVS man page or documentation for a description of the options.) The `repository` argument specifies the directory where CVS will store the project files *within* the repository. The vendor and release tags specify vendor and release information.

Let's import a project into CVS for use under Project Builder:

1. Create a new Cocoa project called `CocoaExample` and store it in your project directory. Build and close the project.
2. Open the Terminal application and change directory (`cd`) to the directory that holds the project:

   ```
   % cd ~/projects/CocoaExample
   ```

3. Check in the project with the `import` command. The `–m` option is used to enter your revision control message from the command line. If you remove it, CVS will open the editor specified in the `CVSEDITOR` variable. When CVS imports or checks in files, it displays a file status symbol, expressed as the single, leftmost character (table 3.1 lists the symbols seen at the beginning of the lines and their descriptions). The `import` command is as follows:

   ```
   % cvs import -m "Initial revision." projects/CocoaExample book start
   N projects/CocoaExample/main.m
   cvs import: Importing /Users/omalley/cvs-repository/projects/
     CocoaExample/CocoaExample.pbproj
   N projects/CocoaExample/CocoaExample.pbproj/omalley.pbxuser
   N projects/CocoaExample/CocoaExample.pbproj/project.pbxproj
   cvs import: Importing /Users/omalley/cvs-repository/
   projects/CocoaExample/English.lproj
   N projects/CocoaExample/English.lproj/InfoPlist.strings
   cvs import: Importing /Users/omalley/cvs-repository/
   projects/CocoaExample/English.lproj/MainMenu.nib
   N projects /CocoaExample/English.lproj/MainMenu.nib/classes.nib
   N projects /CocoaExample/English.lproj/MainMenu.nib/info.nib
   N projects /CocoaExample/English.lproj/MainMenu.nib/objects.nib

   No conflicts created by this import
   ```

Table 3.1 CVS status commands

Symbol	Description
A	File added
C	File merged, changes
E	File exported
F	File released
G	File merge successful
M	File modified
N	New file added
O	File checked out
R	File removed
T	Tag
U	File exists in repository; new revision created
W	File removed from entries file

4 Change directory to the parent of `CocoaExample` (`cd ~/projects`) and remove the `CocoaExample` directory (`rm -rf CocoaExample`). Change directory to the parent of the project directory and check out the version (do not remove or mess with the CVS directory; CVS uses it to resolve differences between local versions of files and those stored under version control):

```
% cd ~/projects; rm -rf CocoaExample; cd ..
% cvs co projects/CocoaExample
cvs checkout: Updating projects/CocoaExample
U projects/CocoaExample/main.m
cvs checkout: Updating projects/CocoaExample/CocoaExample.pbproj
U projects/CocoaExample/CocoaExample.pbproj/omalley.pbxuser
U projects/CocoaExample/CocoaExample.pbproj/project.pbxproj
cvs checkout: Updating projects/CocoaExample/English.lproj
U projects/CocoaExample/English.lproj/InfoPlist.strings
cvs checkout: Updating projects/CocoaExample/English.lproj/MainMenu.nib
U projects/CocoaExample/English.lproj/MainMenu.nib/classes.nib
U projects/CocoaExample/English.lproj/MainMenu.nib/info.nib
U projects/CocoaExample/English.lproj/MainMenu.nib/objects.nib
```

Within the `CocoaExample` directory, you will see a directory called CVS:

```
% cd projects/CocoaExample; ls
CVS  CocoaExample.pbproj  English.lproj  build  main.m
```

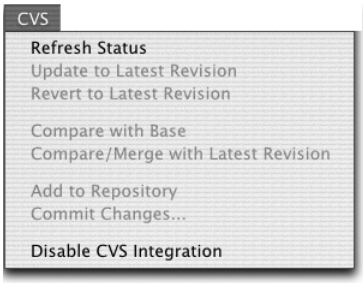

Figure 3.21
Project Builder enables the CVS menu items when you open a project that was previously checked into CVS.

5 Launch Project Builder and open the `CocoaExample` project. You should see a CVS icon at the top left of the Contents pane; it indicates that Project Builder recognizes the project is under version control. Project Builder will also enable the CVS menu when you save changes to a source file (see figure 3.21).

You can use these menu items to interact with CVS and perform operations on the repository. Not all of the CVS commands are available from the CVS menu, but most of the basic ones are there. Typically, you will interact with the CVS repository from the command line and from within Project Builder. If you are a purist, you can still use CVS from the command line only.

Creating new targets and build styles

As you have already seen, a target collects all necessary components that make up a project, as well as instructions for building the project. Components include source files, header files, and libraries. On the other hand, build styles enable you to customize the build options of a specified target. Let's look more at the relationship between targets and build styles through an example.

Imagine you are developing an application that implements two Fibonacci number generators, recursive and loop-based, and you wish to see how the different compiler optimizations affect performance. By using targets and build styles, you can compare two implementations without changing any client code in the main function. Follow these steps:

1 Launch Project Builder and create a new project using the C++ Tool template.

2 Create and add two files to the project, one for each implementation. Call the files FibonacciRecursive.cpp and FibonacciLoop.cpp. Implement each algorithm in its corresponding file and add code to `main` to call the function. Listing 3.1 shows the code for the different implementation of the Fibonacci program and the program's `main` function.

Listing 3.1 Two implementations of the Fibonacci program and the main function

```
/*
 Loop (iterative) implementation of the Fibonacci number
 series generator.
 */
long
Fibonacci(long n)
{
  if (n == 0)
    return 0;

  long x = 1;
  long y = 0;
  long z = 0;
  for (long i=1; i<n; i++) {
    z = x;
    x += y;
    y = z;
  }
  return x;
}

/*
 Recursive implementation of the Fibonacci number series generator.
 */
long
Fibonacci(long n)
{
  if (n == 0)
    return 0;

  if ( (n == 1) || (n == 2) )
    return 1;

  return Fibonacci(n-1) + Fibonacci(n-2);
}

using namespace std;
#include <iostream>

#include "FibonacciRecursive.h"
#include "FibonacciLoop.h"

const int N_FIBONACCI_NUMS = 10;

int
main()
{
  long fibonacciResult[N_FIBONACCI_NUMS];

  for(int i=0; i<N_FIBONACCI_NUMS; i++) {
    fibonacciResult[i] = Fibonacci(i);
  }
```

```
    cout << "Computed Fibonacci values:" << endl;
    for(int i=0; i<N_FIBONACCI_NUMS; i++) {
      cout << "fib(" << i << ") = "
           << fibonacciResult[i] << endl;
    }
    return 0;
}
```

3. Now that the code is in place, you need to create two targets for testing. To create a new target, select Project→New Target or click on the Targets tab from the Contents pane, Control-click (right-click) on the upper pane, and select New Target. Select Tool and click on the Next button; name the new target Recursive.
4. Create another target, called Loop.
5. Add the appropriate source code files to its target (see figure 3.22). Activate the recursive target by clicking the radio button to its left. Click on the Files tab in the Contents pane, and click the checkbox to the left of the files main.c, FibonacciRecursive.cpp, FibonacciRecursive.h, and libstdc++.a (located under the External Frameworks and Libraries).
6. Activate the loop target and do the same, this time selecting main.c, FibonacciLoop.cpp, FibonacciLoop.h, and libstdc++.a.
7. To test each implementation, select its target and build and run the program.

As you can see, targets are a convenient way to build versions of a program that contain different files.

Next, let's look at creating and applying build styles to the project. Build styles enable you to override aspects of a target's build settings. By default, two build styles are defined: Development and Deployment. Each style overrides the build settings of the active target. Let's create a few build styles for testing the effect of different compiler optimizations.

To create a new build style, follow these steps:

1. Select Project→New Build Style or click on the Targets tab from the Contents pane, Control-click (right-click) on the lower pane, and select New Build Style from the menu.
2. Enter the name of the build style. Within the Fibonacci project, create four build styles: OptimizationNone, Optimization1, Optimization2, and OptimizationSize.

```
Completed phase <RezResourceManagerFiles> for Recursive

Standard Logs:
/usr/bin/jam -d1
JAMBASE=/Developer/Makefiles/pbx_jamfiles/ProjectBuilderJambase
JAMFILE=- build ACTION=build TARGETNAME=Recursive NATIVE_ARCH=ppc
BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/omalley/projects/TargetBuildExample/
build/intermediates/Recursive.build/Headermaps/Recursive.hmap
DSTROOT=/ OBJROOT=/Users/omalley/projects/TargetBuildExample/
build/intermediates SRCROOT=/Users/omalley/projects/
TargetBuildExample
SYMROOT=/Users/omalley/projects/TargetBuildExample/build
...updating 11 target(s)...
BuildPhase Recursive
Completed phase <CopyHeaders> for Recursive
Mkdir /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc
CompileCplusplus /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/main.o
CompileCplusplus /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/FibonacciRecursize.o
BuildPhase Recursive
Completed phase <DeriveAndCompileSources> for Recursive
MasterObjectFile.Combine /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/master.o
StandaloneExecutable /Users/omalley/projects/TargetBuildExample/build/
  Recursive
BuildPhase Recursive
Completed phase <LinkWithFrameworksAndLibraries> for Recursive
BuildPhase Recursive
Completed phase <RezResourceManagerFiles> for Recursive
...updated 11 target(s)...

Detailed Logs:
/usr/bin/jam -d2 JAMBASE=/Developer/Makefiles/pbx_jamfiles/
  ProjectBuilderJambase
JAMFILE=- build ACTION=build TARGETNAME=Recursive
NATIVE_ARCH=ppc BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/omalley/projects/TargetBuildExample
/build/intermediates/Recursive.build/Headermaps/Recursive.hmap
DSTROOT=/ OBJROOT=/Users/omalley/projects/TargetBuildExample/
build/intermediates SRCROOT=/Users/omalley/projects/
TargetBuildExample
SYMROOT=/Users/omalley/projects/TargetBuildExample/build
...updating 11 target(s)...
BuildPhase Recursive

    echo  Completed phase "<CopyHeaders>"  for "Recursive"
```

```
Completed phase <CopyHeaders> for Recursive
Mkdir /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc

    /bin/mkdir  -p "/Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc"

CompileCplusplus /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/main.o

    /usr/bin/cc  -c "-F/Users/omalley/projects/
TargetBuildExample/build"  "-I/Users/omalley/projects/TargetBuildExample/
  build/include"
  "-arch" "ppc" "-fno-common" "-fpascal-strings" "-O0"
"-Wmost" "-Wno-four-char-constants" "-Wno-unknown-pragmas"
"-pipe" "-precomp-trustfile" "/Users/omalley/projects/TargetBuildExample/
  build/
intermediates/Recursive.build/TrustedPrecomps.txt"
"-Wp,-header-mapfile,/Users/omalley/projects/
TargetBuildExample/
build/intermediates/Recursive.build/Headermaps/
Recursive.hmap"   "-I/Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/DerivedSources"
"main.cpp"  -o "/Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/main.o"

CompileCplusplus /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/FibonacciRecursize.o

    /usr/bin/cc  -c "-F/Users/omalley/projects/
TargetBuildExample/build"  "-I/Users/omalley/projects/TargetBuildExample/
  build/include"
"-arch" "ppc" "-fno-common" "-fpascal-strings" "-O0"
"-Wmost" "-Wno-four-char-constants" "-Wno-unknown-pragmas"
"-pipe" "-precomp-trustfile" "/Users/omalley/projects/TargetBuildExample/
  build/
intermediates/Recursive.build/TrustedPrecomps.txt" "-Wp,
-header-mapfile,/Users/omalley/projects/TargetBuildExample/
build/intermediates/Recursive.build/Headermaps/Recursive.hmap"
"-I/Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/DerivedSources"
"FibonacciRecursize.cpp"  -o
"/Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/FibonacciRecursize.o"

BuildPhase Recursive

    echo  Completed phase "<DeriveAndCompileSources>"
for "Recursive"
```

```
Completed phase <DeriveAndCompileSources> for Recursive
ClearFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/LinkFileListPrelink

/bin/rm  -rf "/Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/LinkFileListPrelink"

AppendToFileList /Users/omalley/projects/TargetBuildExample/build/
  intermediates/
Recursive.build/Objects/LinkFileListPrelink

    for file_reference in
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/ppc/main.o"
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/ppc/FibonacciRecursize.o"
    do
        echo  "$file_reference" >>
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/LinkFileListPrelink"
    done

MasterObjectFile.Combine
/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/master.o

    /usr/bin/cc  -arch ppc  -keep_private_externs   -nostdlib
-filelist "/Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileListPrelink"  -r -o
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/master.o"

ClearFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileList

    /bin/rm  -rf
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/LinkFileList"

AppendToFileList
/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/LinkFileList

    for file_reference in
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/master.o"
    do
        echo  "$file_reference" >>
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/LinkFileList"
    done

StandaloneExecutable
```

```
/Users/omalley/projects/TargetBuildExample/build/Recursive


StandaloneExecutable.LinkUsingFileList
/Users/omalley/projects/TargetBuildExample/build/Recursive

    /usr/bin/cc  -o "/Users/omalley/projects/TargetBuildExample/build/
  Recursive"  "-
L/Users/omalley/projects/TargetBuildExample/build" "-
L/usr/lib/gcc/darwin/2.95.2"  "-
F/Users/omalley/projects/TargetBuildExample/build"  -filelist
"/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/Objects/LinkFileList"  "-arch" "ppc" "-prebind"
"-lstdc++"

BuildPhase Recursive

    echo  Completed phase "<LinkWithFrameworksAndLibraries>"
for "Recursive"

Completed phase <LinkWithFrameworksAndLibraries> for Recursive
BuildPhase Recursive

    echo  Completed phase "<RezResourceManagerFiles>"
for "Recursive"

Completed phase <RezResourceManagerFiles> for Recursive

...updated 11 target(s)...
```

The minimal setting only shows the basics of the build: the commands run and any errors and warnings. The standard setting provides more information about each step in the build process. The detailed setting shows the commands run, their command line, and warnings and errors. Note the use of the Jam program (a `make` replacement) for managing the build process; it also uses different debugging values (`-d0`, `-d1`, `-d2`).

Now, let's look at the Compiler Settings section in the Editor pane. The Code Generation area is used to set the desired optimization level for the build. The optimization levels in the menu correspond to the standard `gcc` optimization levels (see table 3.2).

Table 3.2 The compiler optimization levels avaliable under `gcc`/`g++`

Menu item	`gcc` option	Description
None (less optimized, more debuggable)	-O	Does not optimize
Level 1 -O1	-O0	Optimizes

Table 3.2 The compiler optimization levels avaliable under `gcc`/`g++` *(continued)*

Menu item	`gcc` option	Description
Level 2 -O2	-O2	Performs most supported optimizations except loop unrolling, function inlining, and register renaming
Level 3 (more optimized, less debuggable)	-O3	Turns on all optimizations
Optimize for size	-Os	Turn on optimizations that do not increase program size.

(See the `gcc` documentation's section "Options That Control Optimization" for more specific information about optimizations settings.)

The next option is the Generate Profiling Code checkbox. Enabling this box adds `-pg` to your build options, which adds code to support program performance analysis with `gprof`. The `gprof` program is used to display a performance execution profile for your program.

Enabling the Generate Debugging Symbols checkbox adds the `-g` option to the build, which adds symbolic information to the object files, enabling `gdb` to provide you with more information while debugging. The Other C Compiler Flags text field is used to adding additional compiler flags to the build.

Link options are set in the Linker Settings portion of the Build Settings section. This section enables you to customize elements of the link phase of the build. Project Builder uses `ld`, the Mach object file link editor, to perform link operations; `libtool` to create static and dynamic libraries; and `dyld` to load an application's dynamic link libraries into its address space.

Development from the command line under Project Builder

In addition to building your programs from within Project Builder, you can choose to build them from the command line using the `pbxbuild` command. To use the command, open a shell and change to the directory that contains your project. The `pbxbuild` program has several command-line options:

```
pbxbuild [-activetarget | -alltargets | -target <targetname> ]
[-buildstyle <stylename>] [ clean | install ]
[ <variable>=<value> ]
```

To build Project Builder active target, use the `-activetarget` option; to build all targets in the project, use `-alltargets`; or to build a specific target, use the `-target` option, followed by the target name. You apply build styles by specifying the build style option followed by the name of the style. Both target and build style names

are case sensitive. In addition, `pbxbuild` supports make-like options such as `clean` and `install`, which build and install the program in the target directory.

The makefile in listing 3.3 is a simple example of how to use `pbxbuild` and its options.

Listing 3.3 Makefile for building a project from the command line using pbxbuild

```
#------------------------------------------------------------------
#  $Id$
#------------------------------------------------------------------

BUILD_TOOL = pbxbuild

# Uncomment and edit <stylename> to a Project Builder
# build style name.
BUILD_STYLE = # -buildstyle <stylename>

# ------------------------------------------------------------
# Build targets
# ------------------------------------------------------------
all:
        $(BUILD_TOOL) -alltargets $(BUILD_STYLE)

active:
        $(BUILD_TOOL) -activetarget $(BUILD_STYLE)

# Edit <targetname> to target to build.
target:
        $(BUILD_TOOL) -target <targetname> $(BUILD_STYLE)

# ------------------------------------------------------------
# Build action (can specify more than one):
#  export install clean installsrc
# ------------------------------------------------------------
clean:
        $(BUILD_TOOL) clean

# Must specify SRCROOT in environment.
export:
        $(BUILD_TOOL) SRCROOT=. export

install:
        $(BUILD_TOOL) install

list:
        $(BUILD_TOOL) -list
```

You can run this makefile from the command line or from within an editor like `emacs`. The advantage of running it within an editor is that you can use the editor's next/previous message command to jump to the source line of an errors or warning. (Within `emacs`, use Control-x-~ to get the next compiler error or warning.)

Adding static checking to builds

Static code analysis refers to techniques and methods applied before a program is run that highlight potential problems, anomalies, or errors in source code. In the software engineering literature, as well as in practice, static code analysis means different things to different people, and consists of many techniques and methods. These include peer and formal review sessions, formal methods, software metrics, and methods that focus on detecting language-based and programming problems. In spite of the varying methods, the goal is the same—to examine a program's source code using some measurable procedure with the goal of detecting and removing prospective errors.

Historically, developers of early C compilers made a clear separation between static analysis and compilation. In the spirit of UNIX design, a program should do one thing and do it well. In this spirit, compiler writers designed their compilers to be as small and fast as possible, leaving static analysis to another program, called `lint`. Some feel this approach was a mistake. As Peter van der Linden points out, many programmers do not use `lint` for semantic analysis, so we get faster compilation, but at the cost of allowing many detectable bugs to get past the compiler.[4] Today, most compiler vendors implement stricter semantic checking in their compilers. For example, `gcc` and `g++` provide a wide range of options for detecting semantic errors in source code, and Sun's CC compilercontains options that are even more advanced.

One of the easiest and more productive static code analysis techniques is to use your compiler's warning flags to detect programming errors. During the development process, your first line of defense is compiler options. By intelligently using compiler options, you can use the compiler to alert you to potential problems in your source code early in the development process.

To use Project Builder for semantic code analysis of C and C++ code, you need to understand `gcc`'s compiler flags. By enabling these flags, you tell the compiler to perform stricter semantic checking when processing source code. The `gcc` manual groups warnings into the following categories:

- Warning options
- C language options
- C++ language options

[4] Peter van der Linden. *Expert C Programming!* (Englewood Cliffs, N.J.: SunSoft Press,1994). 50–60. He also provides an interesting account of Sun's use of `lint` for its kernel code.

Warning messages tell the compiler to check for language or programming constructs that are potentially dangerous or may lead to errors or unexpected results. This is one of the most useful sets of options supplied by the compiler. Both C and C++ language options define a set of options that detect and verify conformance with various dialects of C, C++, and Objective C. These options are useful if you wish to check your code for conformance to a particular language standard.

For example, the `–Wall` option collects many useful compiler flags under a single switch, and the `–W` option adds even more checking. Including these two options in your build is a great way to perform basic semantic code analysis.

To set compiler warning flags within Project Builder, follow these steps:

1. Select a target and click on the Build Settings tab of the Edit pane (changed in Mac OS X 10.2 to a hierarchical list of settings with subpanes in Expert view).
2. In the list of build settings, double-click on the value section of the `WARNING_CFLAGS` record (see figure 3.25). Use this field to add any additional warning compiler flags to the build. Or, under the Compiler Setting section of the Edit pane, use the Other C Compiler Flags text field to add compiler flags.

3.4 Creating an application with Interface Builder

The cornerstone of developing Mac OS X programs using the Apple development tools is Project Builder. You use the Project Builder environment to write your program's source code and build, run, and debug your program. However, for developing GUI-based applications under Mac OS X, this is only half the story. In addition to implementing the program's logic, you also need to create its user interface. Enter Interface Builder.

You use Interface Builder to design the user interface component of your program. The relationship between Project Builder and Interface Builder is similar to that of Project Builder and the UNIX-based development tools. As you know, Project Builder uses the services of the UNIX-based development tools to perform common development tasks. For creating user interfaces, it uses Interface Builder. With Interface Builder, you design application menus, windows, icons, and dialog boxes that provide your application with its GUI.

The best way to understand the components of Interface Builder and its interaction with Project Builder is to see it in action. If you have not already done so, go through section 3.3 to get a feel for how Project Builder works.

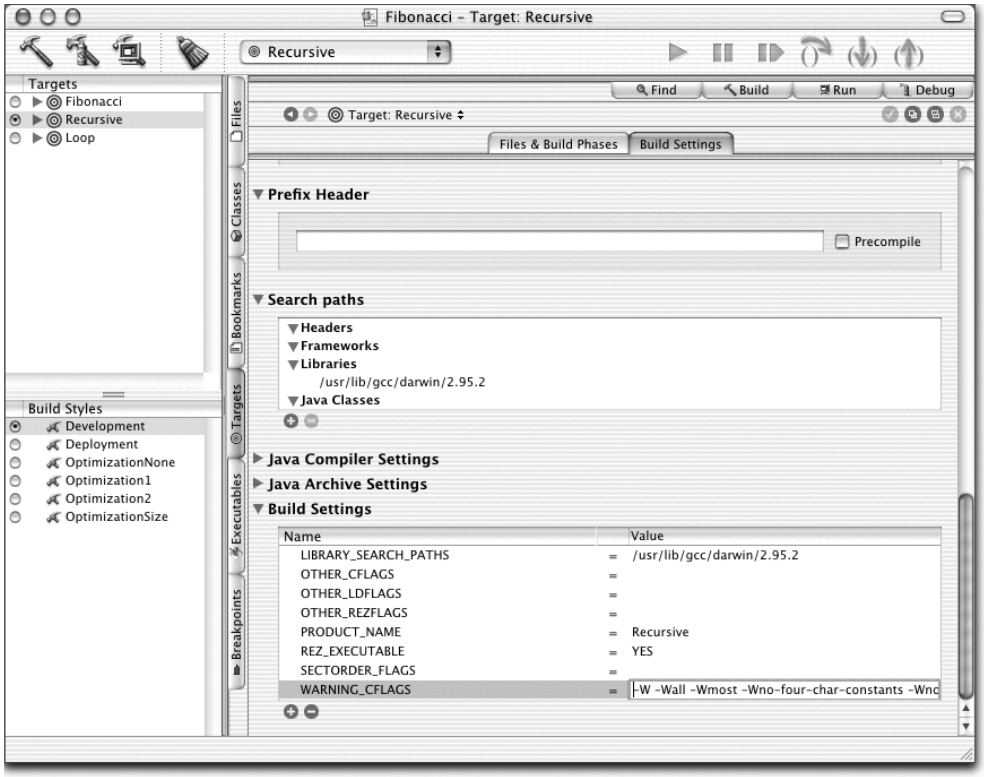

Figure 3.25 The Build Settings category in the Edit pane enables you to set extra compiler flags that Project Builder adds to the build command.

3.4.1 Interface Builder scenarios

The following sections describe typical situations you will encounter when constructing your programs GUI with Interface Builder. These topics will give you a taste for some of Interface Builder's most useful features.

Nib files

Under Mac OS X, you construct a Cocoa application's user interface using Interface Builder and store this information in one or more *Nib* files. Nib files come from the days of NeXT computer and stand for NeXT Interface Builder.[5] Generally, a Nib file holds application interface components.

[5] Aaron Hillegass, *Cocoa Programming for Mac OS X* (Boston: Addison-Wesley, 2002), 12.

For example, the Nib file for a Cocoa program not only contains its user interface components (menus, windows, and so on), but also encodes and stores information about each object and the relationship between these objects within the object hierarchy. The runtime system decodes this information when the program is loaded.

Let's look at the contents of a Nib file using a command-line program called `nibtool`. The `nibtool` program lets you display different information from a Nib file through its command-line options. For example, the –c option displays the local classes in a Nib file, `-j` outputs the setting for the objects, and –x prints connections between the objects. To experiment with this program, open a shell and change to a directory that holds a Nib file (usually under a project's English.lproj directory). Listing 3.4 shows the condensed output of a `nibtool` command.

Listing 3.4 Information from a Nib file, displayed with nibtool

```
% nibtool -c MainMenu.nib
/* Classes */
Classes = {
    IBClasses = (
        {CLASS = FirstResponder; LANGUAGE = ObjC;
                 SUPERCLASS = NSObject; },
        {
            ACTIONS = {clearMe = id; clickMe = id;
                       myMenuAction = id; };
            CLASS = MyClass;
            LANGUAGE = ObjC;
            OUTLETS = {textItem = id; };
            SUPERCLASS = NSObject;
        }
    );
    IBVersion = 1;
}; /* End Classes */

% nibtool -j MainMenu.nib
Objects = {

    "Object 1" = {
        Class = "NSCustomObject";
        CustomClass = "NSApplication";
        Name = "File's Owner";
        className = "NSApplication";
    };

    "Object 2" = {
        Class = "NSView";
        autoresizingMask = "0";
        frameRect = "{{1, 9}, {404, 148}}";
        groupedIBObjectID = "<null>";
```

```
        isLockedIBObject = "0";
    };
% nibtool -x MainMenu.nib
Connections = {
    "Connection 37" = {
        Action = "performMiniaturize:";
        Class = "NSNibControlConnector";
        Source = "23";
    };
    "Connection 39" = {
        Action = "arrangeInFront:";
        Class = "NSNibControlConnector";
        Source = "5";
    };
```

Creating and editing menus, windows, and other interface objects

The usual way to use Interface Builder is in conjunction with Project Builder. Typically, you create a new project within Project Builder and edit its user interface using Interface Builder. From within Project Builder, you double-click on the application's main Nib file, located in the Resource folder, to launch Interface Builder and load the Nib file. At this point, you can edit existing interface components or create addition interface elements.

When you open an application's Nib file in Interface Builder, you will see a window that holds the application's menu (the menu displayed at the top of the screen when the application is running). You can change the text of an existing menu item by double-clicking on its name and editing the text. To add a new menu item, click on the Cocoa Menus item in the Palette toolbar (see figure 3.26), select the item you wish to add, and drag it to its location within the menu window. Dragging it over a menu item opens the menu so you can place the item in the menu. You can add a single menu item by selecting the Item menu item, or choose a predefined menu item from the palette that already contains the menu item.

To delete an item, select it and press the Delete key. Make sure you read the Mac OS X User Interface guidelines to ensure that your application's menus are stylistically correct.

Within Interface Builder, windows and other interface components are easy to construct, customize, and add to your program. For example, to add a new window to your program, simply select the Cocoa Windows item from the palette toolbar and drag it outside of the palette. Doing so creates a new window and adds it to your application instance. Creating other components is just as simple. Even

extended many of the existing TECO macro packages into a single package called Editor MACroS, or `emacs`. Stallman later wrote a new editor, called GNU `emacs`, where the underlying implementation and extension language was a Lisp-based language called elisp.

Mac OS X supports several `emacs` implementations, ranging from the standard terminal-based version that comes with the system, to versions that take advantage of the Mac OS X Aqua interface. The main limitation of the terminal-based version is that it does not display text highlighting or multicolor fonts and does not support the mouse for moving around the screen. However, it's functionally the same `emacs` you get with other UNIX distributions and it integrates well with the BSD command-line development tools. Implementations that are more integrated into the Mac OS X environment include Carbon `emacs`, based on Apple's Darwin port and Andrew Choi's Mac OS port; and `XEmacs 19.14`, which is based on GNU `emacs 18.59` for Macintosh by Marc Parmet and facilitates accessing the Codewarrior development environment over AppleEvents. Don't confuse this `XEmacs` with the `XEmacs` implementation available under most UNIX flavors; the *X* here stands for the *X* in Mac OS X.

Each of these `emacs` versions implements different features. On one end of the spectrum is the terminal-mode implementation. This version is simple and clean, integrates well with the BSD development tools, has a minimal memory requirement (compared to the other versions), and is launched from the command line. The disadvantage is that it is terminal based, does not display text highlighting or multicolor fonts, and does not support mouse interaction. At the other end of the spectrum is `XEmacs` (http://www.porkrind.org/emacs/). This version offers some useful features, including an Aqua interface, application menus, communication with the CodeWarrior development environment over AppleEvents, and text highlighting and multicolor fonts (see the About `emacs` file for more information on its Macintosh-specific features, as well as differences between it and the UNIX version). The disadvantage of this version is that it has a higher memory footprint than the terminal-based version and is not as well integrated with the UNIX-based development tools or environment as the terminal version.

Carbon `emacs` (http://www.porkrind.org/emacs) stands between the terminal-mode implementation and the Aqua-based version. You can run it from either the command line or the Finder, it integrates with the BSD development tools, and it supports text highlighting and multicolor fonts. The disadvantage is a higher memory footprint than the terminal-based version; it also cannot run in the background from the command line. To use it from the command line, add the following statement to your initialization file (.cshrc):

```
alias memacs '[path-to-emacs]/Emacs'

# For example
alias memacs '/Users/omalley/CarbonEmacsEmacs/Emacs.app/
Contents/MacOS/Emacs'
```

New ports of `Emacs` for Mac OS X appear frequently. Watch online forums and announcements for more information.

vi

Another popular editor that runs under UNIX and Mac OS X is `vi`. The `vi` editor, originally written by Bill Joy, combines both line and screen modes within a single editing program. Today, we call the editor `vi`, but technically `vi` is really the visual mode of `ex`, a line-mode editing utility based on `ed.vi`. In a sense, `vi` contains the best of both worlds. In `ex` mode, you get all the power of the command mode editing operations; in `vi`, or visual mode, you get the benefits of screen mode editing (seeing the changes to the text as you make them). Keep in mind that `vi` was created within and for a very specific computing environment. As Bill Joy points out, a design goal was to make `vi` usable over a 300 baud modem. For a screen editor to be usable in this context, commands must be as compact as possible. According to Joy, "People don't know that vi was written for a world that doesn't exist anymore."[1]

A terminal-based `vi` editor (`nvi`) is loaded with the default Mac OS X system. Like the default version of `emacs`, this version does not display text highlighting or multicolor fonts. A native Mac OS X version of `vi`, called `vim` (http://vim.sourceforge.net), is also available. This version extends the functionality of `vi` to include a GUI, split windows, and menus.

Other editors

In addition to the standard UNIX editors, others are available for Mac OS X. These include `joe` (http://tony.lownds.com/macosx), the Wordstar-like editor; and `nedit` (http://www.nedit.org/download/macos.shtml).

If you are interested in getting functionality similar to that of the UNIX implementations, use the terminal-based versions of the editors. Each of the Mac OS X–based versions contains some useful features but require you to make some tradeoffs. In the end, it is best to evaluate each editor and make up your own mind. Another idea is to install X Darwin and a window such as OroborOSX (http://wrench.et.ic.ac.uk/adrian/software/oroborosx), and run `Emacs` and `vi` within X Window. See chapter 2 for more information on installing X Darwin.

---

[1] See http://www.linux-mag.com/1999-11/joy_01.html for the complete interview.

4.2.3 Version control

Version control software facilitates the efficient management of the modification and revision history of files throughout a project's life cycle. Version control is a topic within a discipline of software engineering called Software Configuration Management (SCM). SCM is a mechanism, instituted through defined processes, whose goal is to ensure the classification, control, and traceability of a software system throughout its life cycle; it is implemented using software tools and procedures designed to address these objectives. Configuration management was rooted in the defense industry of the early 1960s, and was an early attempt by management to control the increasing complexity of designs and the design process.

One of the first version control systems available on the UNIX platform was Source Code Control System (SCCS), developed by Marc Rochkind at Bell Telephone Laboratories in 1972. At the present time, the two most popular version control systems are Revision Control System (RCS), written by Walter F. Tichy in the early 1980s while at Purdue University; and Concurrent Versions System (CVS), which is built on top of RCS and uses many RCS programs to perform its actions.

The primary difference between RCS and CVS lies in how they interact with multiple users. RSC locks a file when someone checks it out, which simplifies version control and sidesteps many unnecessary problems. For example, multiple developers editing a file simultaneously can lead to one developer breaking the other's code. Single checkout forces them to talk to each other and resolve any conflicts before changing code.

CVS does not (usually) lock files; instead, it merges changes into each developer's code base. Conflicts can arise, but they are rare. This system allows many people to work in parallel, and as long as they do not create incompatible code, CVS will fold in each developer's changes as requested.

Choosing a version control system

Both CVS and RCS are excellent choices for version control systems. Both are enormously popular, stable, and available under Mac OS X. The following concerns can influence your choice of version control system:

- Will the project have multiple developers, editing and sharing files simultaneously?
- Is the development team geographically distributed? Do members require remote access to files?

Let's look at an example of how a development group might use RCS and CVS on a project. The goal of this project is to develop a compiler. The group is composed

of three developers: A, B, and C. Developer A will work on the front-end of the compiler: the lexical and syntax analyzer and the parser. Developers B and C will implement the back-end of the compiler: developer B writes the code generator, and developer C implements the code optimizer. The project uses shared files (io.c and io.h) that contain common I/O operations.

Scenario 1: The group members decide to use RCS for the project (configured for strict locking, its default behavior). Work progresses as follows:

1. All three developers begin work on their part of the compiler, checking in code as necessary. Assume that io.c and io.h are under version control.
2. Developer A checks out the head version of io.c and io.h (setting the file lock: `co -l files`), adds some functions, and checks in both files, thereby releasing the lock.
3. Developers B and C check out the head versions to get the new changes (without setting the lock: `co io.c, co io.h`).
4. Developer B checks out the head version of io.c and io.h (this time setting the file lock), adds some functions, and goes home for the night. At this point, developer B holds the lock on io.c and io.h.
5. That night developers A and C need a common I/O function to continue with their work. Developer C checks out io.c and io.h, adds the function, and attempts to checks in the files so developer A can use the new function. Unfortunately, RCS rejects this operation because developer B still owns the lock on the files.
6. Development stops until developer B checks in the files, thereby releasing the lock.

This problem can be sidestepped by setting the file-locking mode to nonstrict and making sure all developers own the file, possibly working under the same user account. However, this arrangement is highly unlikely and defeats the primary reason of using version control in the first place. In addition, developers A and C can break the lock on the file by using the `-u` or `-M` option, but this goes against the design intent of RCS.

Scenario 2: The group members decide to use CVS for their project. Work progresses as follows:

1. All three developers begin work on their part of the compiler, checking in code as appropriate. Assume that io.c and io.h are under version control.
2. Developer A checks out the head version of io.c and io.h (`cvs co io.c, cvs co io.h`), adds some functions, and commits both files.

3. Developers B and C perform an update to get the new changes. Developer B adds some functions to io.c and io.h, and goes home for the night.
4. That night, developers A and C need a common I/O function to continue with their work. Developer C performs an update, getting the latest version of io.c and io.h, adds the function, and commits the files so developer A can use the new function. Developer A performs an update, getting the latest version of the files that include the new function. Developer B is at home, completely unaware of the new additions.
5. After returning, developer B performs an update, and CVS merges any new changes into the working version of io.c and io.h. If there are any conflicts, CVS alerts developer B, and changes are manually fixed.

For this use, CVS is clearly the right choice.

To illustrate the second question (a geographically distributed development team), envision the following: you and some friends want to develop a new editor for Mac OS X. Each person lives in a different part of the country. A fundamental requirement of the project is that members need to be able to access and update each other's work at any time. CVS is designed to work over the network, so one developer sets up the CVS server on their machine and sets up a repository. The other developers configure their environment to access the repository remotely over the network. Now all developers have access to CVS as though the repository were accessible within their file system.

Further, suppose you are working in one location and relocate for a few months to another part of the country. You can set up the CVS server on your home machine and, when you get to your new location, set up the new machine as a CVS client, accessing the remote repository over the network. Now, you can retrieve files from your remote repository as if you were on your local machine.

These examples demonstrate the primary differences between RSV and CVS. Because this chapter is about Project Builder, I will focus more on using CVS for version control.

Overall, RCS is a good choice for small projects that do not require developers to simultaneously share and edit files; it is ideal for one-person development projects. It is easy to set up, the command set is straightforward to learn, and it consumes few system resources. Unfortunately, Project Builder, Apple's core IDE for developing Mac OS X applications, does not support RCS. However, this doesn't mean you can't use RCS as a version control system when developing under Project Builder; you just need to access it from the command line.

CVS is an excellent choice for multideveloper projects where simultaneous file sharing and editing is a requirement. In addition, CVS works over a network, and is therefore ideal for projects with geographically distributed developed teams, such as open source projects. Project Builder also supports CVS as its primary version control management tool, so it is the right choice if you plan to develop programs under Project Builder and want to share a single code repository.

Setting up RCS

RCS is very simple to set up. The primary decision is where you want to store RCS files: in the working directory of the project or in a directory within this working directory, called RCS. RCS stores file differences, or *deltas*, in a file called the RCS file. The difference file holds the revision history of the corresponding file in a space-efficient manner.

Each file placed under version control has a parallel RCS file called [filename],v. For example, if you place the file parser.c under version control, the corresponding RCS file is called parser.c,v. If an RCS directory exists within the working directory, RCS will store the RCS file there; otherwise, RCS stores files in the working directory.

Setting up CVS

Setting up CVS takes a few more steps. Before using CVS, you need to configure the CVS repository and the client machine environment:

1. Create a directory called the CVS *repository*, which holds all files stored under version control.
2. Set the CVS environment variable `CVSROOT` to the location of this repository directory (the directory you just created) and the `CVSEDITOR` environment variable to the editor you wish to use to enter revision messages.
3. Run the CVS `init` command to create the CVS administrative files in the repository.

The following example demonstrates the CVS commands you use to set up the environment and create an administrative file in the root repository:

```
% mkdir /cvs-repository
% setenv CVSROOT /cvs-repository
% setenv CVSEDITOR emacs
% cvs init
```

For ease of use, add the environment commands to your initialization file so they are automatically set. Once the version control environments are set up, you can

use them just as you would under UNIX. For more information about RCS and CVS, see their man pages.

4.2.4 Static code analysis tools

UNIX has always been strong in providing high quality developer tools, and static code analysis tools like `lint` are no exception. *Static code analysis* refers to techniques and methods applied before running a program that highlight potential problems, anomalies, or errors in source code. Compiler warning flags offer some protection, but many programmers use `lint` to perform static analysis on their source code.

`Lint`, originally written by Stephen C. Johnson in 1978, arose because the designers of early C compilers made a clear separation between static analysis and compilation. Early compiler writers designed their compilers to be as small and fast as possible, leaving static analysis to another program, called `lint`. Today, compiler vendors and developers are implementing stricter semantic checking in their compilers.

The default load of Mac OS X and the developer tools installs some support for static analysis: `gcc`/`g++` and Perl Lint (B::Lint). By enabling certain `gcc`/`g++` options, you tell the compiler to perform stricter semantic checking when processing source code.

In addition, the open source community has some very good tools that work under Mac OS X, which you can use to detect potential semantic errors in your code. One of the best is Splint (formerly LCLint), available from http://www.splint.org. Splint statically checks C source code for potential coding errors and possible security violations. One of Splint's design goals is to detect many possible programming errors but limit the number of spurious messages, which can be a problem with other `lint` versions. Splint also supports the notion of annotations, which permit you to add comment-based directives to source code to provide Splint with more information about what you really mean, thereby enabling it to detect more errors and skip false positives.

Splint may require a few extra steps to build under Mac OS X. To build Splint:

1. Decompress the distribution:

   ```
   tar zxfv splint-[version].src.tgz
   ```

2. Execute `./configure`.
3. Open config.status, look for file path names split over more than one line (around line 310), make each into a single line, and save the file.
4. Execute `./config`.status (doing so generates correct makefiles).
5. Execute the `make` command.

4.3 Compilers and build tools

The Mac OS X development tools come with all the usual UNIX build tools, including `gcc`, `g++`; supporting programs written in C, C++, Objective-C, and Objective-C++; `gdb`;, `make`; Java; and `as` (GNU assembler). Keep in mind that under Mac OS X, `gcc` is called `cc` and `g++` is called `c++`. This naming convention will cause problems when you try to compile projects that look for the `gcc`, `g++` compiler. The way around this situation is simply to create soft links for each, called `gcc` and `g++`, to maintain compatibility with UNIX naming conventions (you must be user root to create these soft links):

```
% cd /usr/bin
% ln –s /usr/bin/cc gcc; ln –s /usr/bin/c++ g++
% ls -l cc gcc cpp g++
-r-xr-xr-x  1 root  wheel  113692 Dec 21 17:31 cc
-r-xr-xr-x  1 root  wheel    3207 Sep  2 23:23 c++
lrwxr-xr-x  1 root  wheel      12 Dec 21 17:42 g++ -> /usr/bin/c++
lrwxr-xr-x  1 root  wheel      11 Dec 21 17:42 gcc -> /usr/bin/cc
```

4.4 Mac OS X Aqua-based development tools

In addition to the customary UNIX text-based tool set, Mac OS X supports many UNIX programs that developers have ported to the Aqua user interface. This means you can use some of your favorite UNIX programs with new, Aqua-based interfaces.

4.4.1 UNIX-based editors

As you saw in the previous section, all the familiar UNIX editing tools are available under Mac OS X, including notable favorites like `emacs` and `vi`. Within the Mac OS X Aqua environment, you have several choices of non-UNIX, Aqua-based programming editors.

Project Builder editor

Let's begin with the editor that comes with Project Builder. The Project Builder editor provides most of the basic editing features you would expect, as well as some advanced features such as `emacs`-style key-mappings, syntax highlighting, and indentation options. You can customize the editor's behavior through the Project Builder Preferences dialog (located under Project Builder→Preferences) using the Text Editing, Syntax, and Indentation items (see figure 4.2). These options offer most of the common customization features you will require for basic editing tasks, but certainly do not offer the breadth of customization

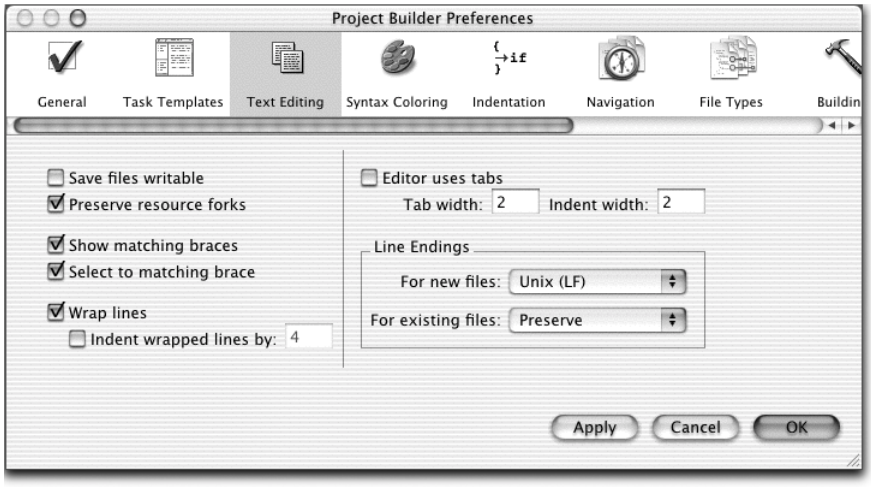

Figure 4.2 The Project Builder Preferences dialog enables you to set many customization options for Project Builder's integrated editor.

options supported by UNIX editors like `emacs`. Let's look at some of the more interesting options.

The Text Editing item enables you to set various editing options that affect how the editor treats, formats, and saves information:

- *Preserve Resource Forks*—Permits you to save files with or without their resource fork. Because many Classic mode programs expect files to have resource forks, this option is useful if you are editing shared files from the Mac OS X and Classic environments.[2]
- *Line Endings*—Useful if you are editing a set of files for a cross-platform project and you need to preserve file formats between platforms. These options are helpful if your code base and primary development environment are UNIX, but you also plan to do development from the same code base under Mac OS X and Project Builder. In this case, you would set the For Existing Files menu item to Preserve, to ensure that Project Builder maintains the UNIX line endings.

The Syntax Coloring pane item permits you to set the font, colors, and styles Project Builder applies to a source file (see figure 4.3):

[2] Not all Classic programs expect a resource fork, but many do. For example, the Codewarrior IDE and the BBEdit text editor use the resource fork for storing font and size information.

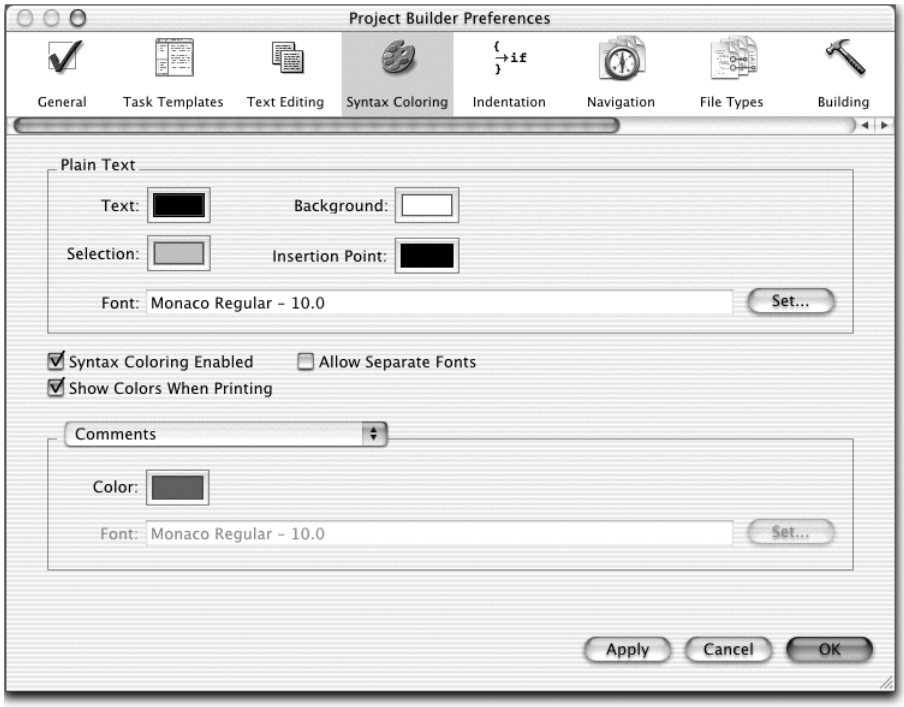

Figure 4.3 You set font color and style options using Project Builder's Syntax Coloring preferences.

- *Allow Separate Fonts*—Enables the editor to display different fonts for different language elements. Imagine you like string constants italicized. You select the Strings item from the pop-up menu (below the checkbox), check the Allow Separate Fonts checkbox to enable the Font text item (located at the bottom of the dialog box), and click Set. Then, select the appropriate font and typeface and click OK. The editor now displays all strings as italicized. Deselecting the checkbox will remove the font and typeface highlighting. Contrast this interface with `emacs` or `vim`, where you specify typeface styles and options in an initialization file. Doing this through an interface makes the job faster, but is not as extendable.
- *Show Colors When Printing*—Prints source code listings with stylized fonts and font types. You can also get this functionality with the UNIX tool `trueprint`, but having the feature available with Project Builder is a real time saver.

Indentation options enable you to specify rules for how the editor indents your code (see figure 4.4). These rules are similar to `emacs` language modes and hooks:

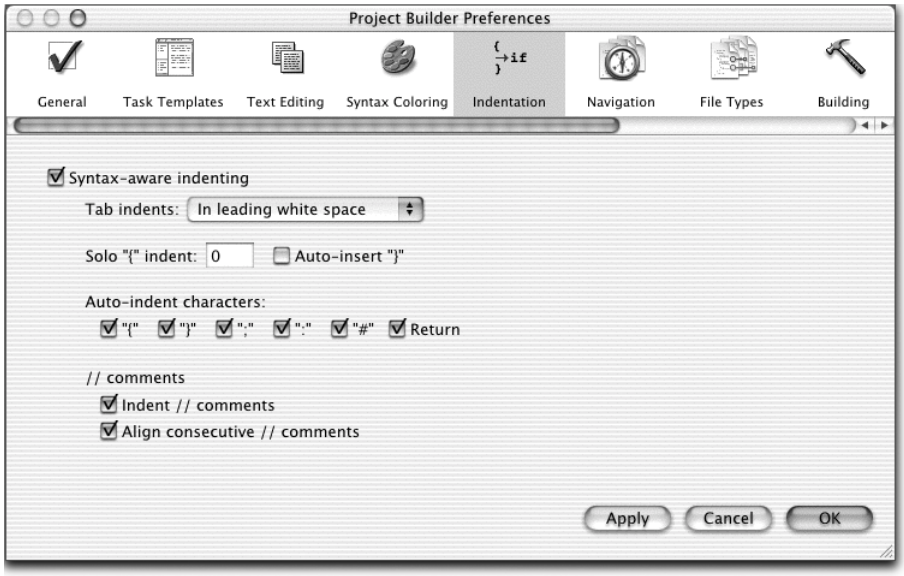

Figure 4.4 Source code indentation options are set under the Indentation preferences. These options are similar in spirit to the options available under most editors such as `emacs` and `vi`.

- *Solo "{" Indent*—Holds the number of spaces the brace is indented.
- *Auto-Insert "}"*—Automatically adds a closing brace after you type an opening brace. If you have ever tried to set up this functionality under `emacs`, you know it is a welcome feature.
- *Auto-Indent Characters*—Indents the corresponding character if it is not entered at the correct indentation level.

Collectively, these editor options provide core functionality and should make you feel right at home within the editing environment.

One interesting consideration is how you know whether applying a set of formatting options to your code has changed the behavior of the program. Granted, the formatting changes Project Builder applies are minor compared to such tools such as GNU indent, but it is still an important question. Intuitively, source code formatting should not alter the operation of a program; it simply reformats code by inserting whitespace into a source file. As Peter van der Linden's excellent book *Expert C Programming* points out, this is not always the case.[3] Nevertheless, how

[3] Peter van der Linden, *Expert C Programming*, 10–11.

do you convince yourself this is true? One technique is to compare the binary file generated from the original source code with the binary file generated from the newly formatted code:

1. Compile the original program (before reformatting), reformat the code, and recompile the program to a different name.
2. Compare the two binaries using the `cmp` command:

```
cmp –l [first-file] [second-file]
```

The `cmp` command should not produce output if the files are the same. This method seems intuitively correct, but under some conditions it may not produce repeatable results. For example, some compilers contain enhancements that randomize the stack layout (insert random values into the stack) at compile time in an attempt to prevent buffer overflow attacks.[4] In this case, there is no guarantee that compiling a program many times will produce the same object code each time.

NOTE Sometimes it is useful to view a binary file in hex. The program `xxd` can be used for this purpose. The following command will produce a text file (in hex) of the binary program:

```
% xxd binary-file > hex-file.txt
```

`emacs` and `vi` can also be used to generate hex files from binary programs.

Another approach is to compile each program to assembly and compare the assembly listings, first using `diff` and then manually if necessary. Yet another approach is to use regression testing to verify that reformatting has not changed the behavior of the program. This technique involves creating a regression test for the program: first the program behavior is baselined by running the regression test; then, after reformatting, you rerun the regression test. Because a regression test is probably already part of the project, no additional coding is necessary to verify reformatting. However, testing is only as sound as your test cases, and providing complete test coverage is a difficult task.

---

[4] StackGuard, now part of Immunix 7.0 (http://immunix.org), is an example of this technology. StackGuard was originally developed under a DARPA-funded Information Survivability research project at the Oregon Graduate Institute of Science & Technology.

External editors

Unfortunately, Project Builder does not currently support using external editors. However, this should not stop you from exploring and using other Mac OS X editing tools for development. You can use any external editor to edit code, and use Project Builder to build and debug the software. If Project Builder detects the file has changed from the version on disk, it will automatically reload the file. This approach has limitations, but if you prefer another editor, it may be worth it.

4.4.2 Mac OS X-based editors

The Macintosh platform has always supported many fine editing tools, and Mac OS X is no exception. One of the most popular Macintosh programming editors is BBEdit, from Bare Bones Software (http://www.barebones.com). BBEdit is a great editor with a loyal following, it has an uncomplicated interface, and it is rock solid. It is available in two versions: the full-featured commercial product and a freeware version that contains a subset of the full version's features. The freeware version does not contain features such as the HTML tools, Unix scripting and command integration, and extensible syntax coloring, to name a few, but it does include the core editing features, making it a very good choice as a free programming editor.

Another popular editor for the Macintosh is Alpha. Alpha is a very good editor that uses Tool Command Language (Tcl) as its extension language. Alpha supports language-based syntax highlighting and many other features for programming and general text editing operations; it's also a favorite editor of users composing LaTeX documents on the Macintosh. The current (as of this writing) public release of Alpha runs only under the Classic environment, but by the time you read this it should be ported to Mac OS X. According to the Alpha developers, the Mac OS X version will add some new functionality, including direct integration with the system's Tcl library, to provide better performance and upgradeability. In addition, it will support seamless editing of files on remote hosts.

4.5 Apple's GUI-based development tools

Along with Project Builder and Interface Builder, the Mac OS X developer tools distribution contains a set of advanced, and very useful, tools to support application development under Mac OS X. These GUI and command-line tools cover a broad range of development areas including runtime memory and thread monitoring, tracing application system calls and usage, performance profiling, class browsing, and interface verification. Collectively, these tools, along with the BSD

commands, provide a solid tool set, giving you all the support you need for effectively developing programs under Mac OS X.

Under Mac OS X, it is important to use these tools during development to verify that your programs are using system resources efficiently. The Mac OS X operating system is structured in layers, from the low-level Mach-based Darwin kernel, through the Quartz graphics layer, to a series of application support layers and frameworks, to the Aqua interface and the application layer where your program runs. As you can imagine, as messages flow from your program's GUI to the lower layers and back again, performance problems can occur. That's why it's important to understand how these layers interact. If you structure your program to take advantage of the BSD core, you will not harm system performance. The development tools will help you investigate these interactions and efficiently pinpoint possible performance bottlenecks and potential errors. The good news for UNIX developers is that the spirit of the UNIX tool set is maintained in these programs, enabling UNIX developers to quickly adjust to the new tools and environment.

Another interesting use of these tools is reverse engineer engineering Mac OS X applications. For example, many of the programs that appear to be self-contained Mac OS X applications are in fact Cocoa interfaces that use the services of UNIX command-line tools. Many of the development tools, as well as the BSD commands, are quite useful in understanding the interaction between the GUI components and the UNIX commands and determining how these programs work.

The remainder of this chapter focuses on the Mac OS X GUI and command-line developer tools installed from the Apple Developer Tools release, showing their features and use during the development cycle.

4.5.1 Apple Help Indexing Tool

The Apple Help Indexing Tool is used to prepare help files for your programs, which are displayed by the Apple Help Viewer. The Apple Help Viewer implements a minimal HTML browser to display HTML-based help files.

The indexing tool's main job is to parse HTML-based documentation files, or *help books*, and create an index file that the Help Viewer uses to efficiently search the help book for information. I discuss using this program to implement online help for your application in chapter 6, when you'll build a functional Cocoa program.

4.5.2 AppleScript Studio

AppleScript Studio is a component of Project Builder that combines four Apple technologies: the AppleScript language, Project Builder, Interface Builder, and the Cocoa application framework. It enables you to place a Cocoa GUI on a program

written in AppleScript. Think if it as a Mac OS X technology that is similar to using the `Tkinter` widget set as an interface for Python scripts.

The advantage of the AppleScript/Cocoa combination over UNIX scripting languages and GUIs is that AppleScript provides access to Mac OS X application services and system function that UNIX-based scripting languages cannot. In addition, the Cocoa interface is more consistent with the look and feel of the Mac OS X environment and provides you with more components for building user interfaces.

AppleScript Studio is available from within Project Builder in two project types: AppleScript applications and AppleScript document-based applications. The AppleScript Studio folder, located within /Developer/Applications, contains example projects and documentation files that demonstrate how to build an AppleScript Studio application. In chapter 7, you will develop a complete AppleScript Studio application. If you prefer script languages to compiled languages, you should definitely look into AppleScript and AppleScript Studio.

4.5.3 FileMerge

You use FileMerge to find differences between files and directories, and also to merge any differences into a new file or directory. At its core, FileMerge is a UNIX `diff` command. In addition to its `diff` services, it offers some other features, including comparing files to a common ancestor and merging files and directories after comparison. Let's look at how FileMerge works and some of its features.

The diff command

FileMerge uses the UNIX `diff` command to perform its basic comparison operations. The `diff` command finds differences between two files, or files within two directories (see the `diff` command's man page for more information). For example, suppose you have two files, fib0.c and fib1.c, and you wish to use the `diff` command to compare them:

```
/* fib0.c */
#include <stdio.h>
long
Fibonacci(long n)
{
  if (n == 0)
    return 0;

  if ( (n == 1) || (n == 2) )
    return 1;

  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

```
/* fib1.c */
#include <stdio.h>
long
Fibonacci(long n)
{
printf("%ld", n);
  if (n == 0)
    return 0;

  if ( (n == 1) || (n == 2) )
    return 1;

  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

The following command compares the two files and displays any differences:

```
% diff fib0.c fib1.c
1c1
< /* fib0.c */
---
> /* fib1.c */
5a6
>       printf("%ld", n);
```

The output displays the differences between the files along with information that shows how to resolve the differences. Let's look at output in more detail.

When `diff` encounters differences between files, it displays the line from each file that does not match, along with a string indicating how to resolve the lines. The less-than (`<`) character indicates that the following line is from the first file; the greater-than (`>`) character indicates that the following line is from the second file. The `diff` command formats this string as

```
[line-number-file-1][action-command] [line-number-file-2]
```

where *`line number`* corresponds to the lines in each file. The action command is an `ed` command, whose meaning is either `a` (append), `i` (insert), `c` (change), `d` (delete line), or `m` move line (`ed` is a line-oriented text editor; see the beginning of this chapter for more information. Therefore, the string `5a6` means that line five of the first file (fib0.c) and line six of the second file (fib1.c) are not the same; to resolve these lines, you need to append (`a`) this line from the second file to the first file.

For our purposes, you also need to know about the `–e` option. It produces output that can be used by `ed` to reconcile the two files. For example, the following command converts fib0.c to fib1.c, printing the result to standard output:

```
% (diff -e fib0.c fib1.c; echo '1,$p') | ed - fib0.c
```

How FileMerge uses diff

Next, let's look at FileMerge and see how it works and how it uses the `diff` command. First, you need to know which UNIX commands FileMerge uses to compare files. A simple way to accomplish this is as follows:

1. Write the following Perl script, which repeatedly looks at the process table for the token `diff`. The problem with this approach is the low resolution at which it acquires process table information, but for this example, it will suffice:

```
#!/usr/bin/perl
# ProcessWatcher.pl
for(;;) {
  system("ps aux | grep /usr/bin/diff | grep –v grep");
}
```

2. Create two large files that `diff` will take a few seconds to process. Doing so enables you to catch the `diff` call in the process table. Remember, you are not interested in the result of `diff`—just that it takes some time to process; the files can contain any information you like.
3. Open two shells, one for running the Perl script and one for killing the Perl script once `diff` has completed. In one shell, run the Perl script (`% perl ProcessWatcher.pl`). In the other, get the script's process identifier (`ps aux | grep ProcessWatcher.pl | grep –v grep`).
4. Open FileMerge, load the two large files you created, and run the compare. Depending on the size of the files, the `diff` operation takes little time to run, but formatting the files within FileMerge can take some time.
5. Once the ProcessWatcher.pl script displays the result of the `diff` command, kill the script (`kill –9 [pid-of-ProcessWatcher.pl]`).

Here is one line of output from the Perl script (edited for readability):

```
/usr/bin/diff -ea 1.txt 2.txt
```

As you can see, FileMerge called the `diff` command with two command-line options. The `–e` option produces output formatted as an `ed` script. The `–a` option tells `diff` to treat the input files as text and compare them line by line. So, the output of this command is an `ed` script that tells `ed` how to resolve and merge the differences in the files. The `diff` program uses the output to show the differences between the files and, if necessary, merge them into a single file. In spite of the fact that this approach is limited to tracking an application's call usage, it works well for simple cases and is easy to implement.

FileMerge features

Let's look at some of the features of the FileMerge program. In addition to comparing files, you can compare and merge all files within two directories.

Another useful feature is the Filter option, located in the Preferences dialog box, which enables you to apply a program to the files you are comparing before they are evaluated. Some predefined filters are available or you can write your own. For example, imagine you wish to `diff` comments from two source files but exclude any code. You can write a program to accomplish this (or better yet, use UNIX commands) and apply it to the files before FileMerge compares the programs.

Ancestor files permit FileMerge to intelligently resolve file differences in creating merged versions of files. If two people begin with the same file (a common ancestor) and make independent modification to each version, FileMerge can use the extra information from the ancestor of both files to make better choices when merging the differences.

The best way to lean about FileMerge's other features is to fire it up and begin using it in your work.

4.5.4 Icon Composer

Users launch a Mac OS X Aqua application by double-clicking on the application icon. As people use your application, they will inevitably begin to associate the application with its icon, so it's important for your application's icon to be as simple and mnemonic as possible. Apple bundles an icon creation program called Icon Composer with its development tools.

To make a set of application icons with Icon Composer, you create your icons, save them in graphics files, import them into Icon Composer, and save the Icon Composer file as an .icon file. You must be aware of a few caveats before you begin:

- Icon Composer imports files stored as either PICT or TIFF files.
- You can create icon files in the following sizes: 16x16, 32x32, 48x48, and 128x128.
- Each icon file must contain an alpha mask to handle transparency.

Many graphics programs are available for creating icon files, but one of the best is Graphic Converter. In chapter 6, you will see an example of how to construct application icons for Cocoa applications.

4.5.5 Interface Builder

Developing a program's user interface is a fundamental task when writing Mac OS X Aqua programs. Under Mac OS X, you create user interface components

using Apple's Interface Builder. The Interface Builder application works hand in hand with Project Builder to develop Mac OS X GUI-based applications. With Interface Builder, you design user interfaces for your program, including application menus, windows, icons, and dialog boxes.

4.5.6 JavaBrowser

The JavaBrowser application enables you to view Java class documentation. The browser is laid out with the upper window using the familiar Mac OS X column browser interface and the lower part holding selected documentation files (see figure 4.5). You can view class documentation by clicking on the various entries and maneuvering between class items.

In addition to viewing documentation, you can search for specific information such as class, method, or field names and view documentation for the result of the search. The documentation provided is terse and of limited use. JavaBrowser can show standard javadocs for Java classes if you click the book icon.

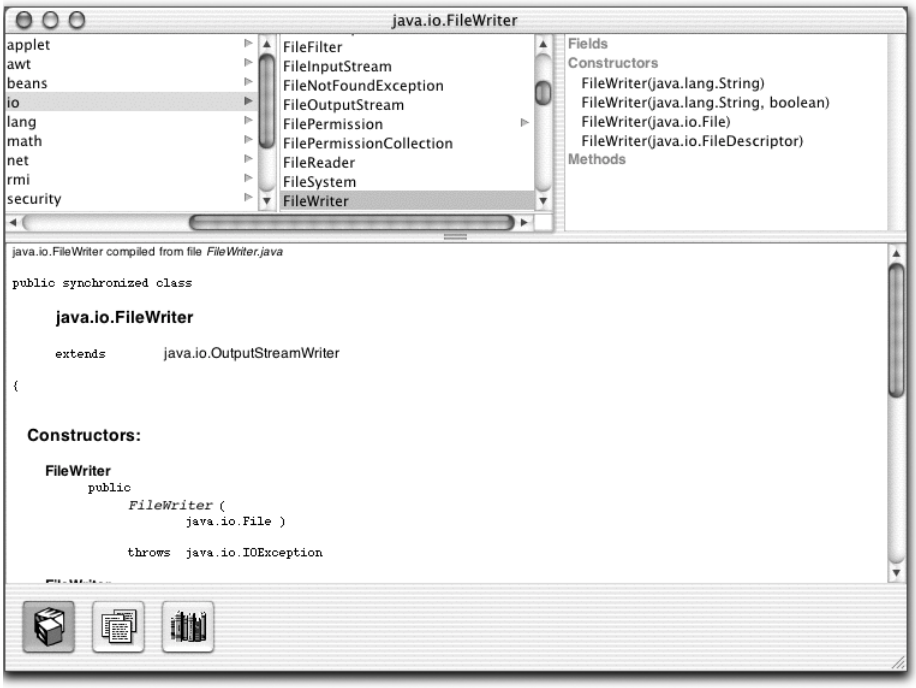

Figure 4.5 The JavaBrowser program displays Java documentation files for selected class, methods, or field names.

4.5.7 MRJAppBuilder

Imagine you just created the next killer application written in Java for Mac OS X and you wish to get it to as many Macintosh users as possible. Because most Macintosh users prefer to launch applications from the Aqua interface rather than the command line, you need a way to make your program available in such a format. Enter MRJAppBuilder (see figure 4.6), a tool Apple provides for creating double-clickable, bundle-based Java applications from *JAR* files (a Java Archive file, which holds all files that compose a Java program within one compressed file).

To create a Mac OS X double-clickable application, you add the .jar file that contains the main class to the Main Classname text field, set the output file name in the Output File text field, and add any other .jar files that compose the application using the Files To Merge Into The Application feature (located under the Merge Files tab). Once you add the files, click the Build Application button and let MRJAppBuilder do its stuff. The result is a double-clickable Mac OS X application that you can distribute to users.

Figure 4.6 MRJAppBuilder lets developers create double-clickable programs from JAR files.

4.5.8 MallocDebug

Programming in languages such as C and C++ provides programmers with lots of power. However, this power comes at a price. In C and C++, one of the biggest costs is that the developer must keep track of all dynamic memory used in a program and make sure the memory is deallocated correctly when it is no longer needed. In theory, this process sounds simple; but in practice, it can be tricky to get right, especially as programs grow in size. Other programming languages, like Java and LISP, address this limitation by implementing garbage collectors, which track memory allocations and reclaim memory when needed.

You can track an application's runtime memory usage manually, or programmatically using specialized libraries that you add at compile or runtime. These libraries replace the default allocation routines with custom calls. At runtime, the program calls the new allocation routines, which store additional diagnostic information, monitor the execution of the program, and report any potential runtime problems such as stack-based errors and memory leaks. (You can also use static analysis tools such as `Pslint` to check memory allocation at compile time.)

The Apple developer tools come with a powerful program called MallocDebug, which helps you detect memory-related errors in your programs. Let's briefly look at memory allocation before getting into the details of how to use MallocDebug.

Memory allocator overview

Computer programs are dynamic entities. As they run, they can require extra storage for holding dynamic data structures that cannot be determined at compile time. This is especially true of object systems that use dynamic binding mechanisms. Programs make requests for extra memory, called *dynamic memory allocation*, through a defined programmatic interface. These memory requests are made through a *memory allocator*. A main goal of the memory allocator is to efficiently allocate and deallocate memory for a program while balancing allocation time versus space tradeoffs.

In C, you accomplish dynamic memory allocation through the `malloc`/`free` family of function calls. Sometimes, the default allocator that comes with your development environment is not sufficient for your needs. In these cases, programmers develop their own versions that replace the default allocator with versions that offer better performance or more features. Implementations can vary greatly, but allocators that come with development environments are usually sufficient for most purposes.[5]

[5] For more information about different allocators and implementations, see "Dynamic Storage Allocation: A Survey and Critical Review" (http://citeseer.nj.nec.com/wilson95dynamic.html) and http://g.oswego.edu/dl/html/malloc.html.

Debugging memory errors

Over the years, programmers have developed many tools and techniques to help C and C++ developers efficiently detect memory-related errors. In the simplest case, the tools non-invasively monitor a program at runtime by watching its overall memory usage. Other tools permit detailed investigation by inserting instructions into the object code that gather statistics about the program's runtime memory behavior. Using these tools, you can get in-depth information about a program's memory usage and whether it's leaking memory or performing any illegal memory operations such as illegal memory accesses, duplicate frees, or buffer overwrites.

In the simplest case, you can perform non-invasive dynamic program analysis on a shoestring by using standard UNIX tools combined with a scripting language. The `ps` command displays what processes are currently running and provides extended information about each process. The `top` command is similar to `ps` but iteratively shows system usage statistics for processes. By controlling either of these commands with a script, you have a simple and easy-to-implement tool for monitoring the runtime behavior of a program. For example, using a Perl script to repeatedly call `ps` for a specific process and outputting its current memory usage enables you to see if the program's memory usage increases over time. Sometimes, this is all that is necessary for you to determine whether a problem exists. The trouble is, this technique does not provide any information about the source of the error within the program or the nature of the problem.

More specialized memory analysis tools provide detailed information about possible errors. Fundamentally, these tools share a common technique: replacing the C/C++ memory allocation and deallocation functions with specialized code that performs extra tracking of allocations and reports any errors. In the most common implementation, each new allocation function allocates additional memory and tags it with specific information. For example, the new allocation code stores a few bytes of information before the allocated block that locates the memory request within the program. It also places a defined byte pattern after the block. At any point when the program is running, or when this memory block is deallocated, the library code checks the trailing block to see if the pattern is preserved. If the pattern does not appear, the code knows a memory overwrite has taken place and uses the leading block information to pinpoint the error.

Memory errors

Now, let's look at some common classes of memory errors in C and C++ programs and how you can detect them with the MallocDebug program. The program

BuggyServer (located in the chapter 4 directory of the book's source code distribution) shows some classes of memory errors that you will detect with MallocDebug.

Open the BuggyServer project in Project Builder by opening its folder and double-clicking the project file, BuggyServer.pbproj. The program is a simple iterative server (after Stevens[6]) that accepts a command, performs an action, and returns a reply to the client. The project README file lists the legal commands you can send to the server. Also included is a Perl script that sends commands to the server, reads the reply from the server, and prints the result. You invoke the script as follows:

```
# send [iterations] [sleep between sends (secs)] [server]
# [port] [message]
% perl send.pl 10 1 localhost 4444 leak
```

This example sends the `leak` command 10 times to the server running on localhost, port 4444, delaying 1 second between sends. Take a quick look through the code that handles the commands, located in BuggyCode.cpp. Each command generates a different class of memory error.

Before we look at some common errors, run the program a few times to get a feel for how its works. To run BuggyServer, press Command-R (Build and Run) or click the Build and Run icon on the toolbar. You should see a message indicating that the server is running on port 4444, as well as the server's process identifier (pid). The server is ready to accept messages. To send the server some messages, open the Terminal application, change to the directory that contains the send script, and enter and execute the following command:

```
% perl send.pl 1 1 localhost 4444 leak
pass: 0
sending: leak:
received: Thu Feb 14 08:08:59 2002
```

This output shows the client sent a `leak` command and the server returned the time it received the request. Also look at the output pane within Project Builder. You should see a log message indicating the time the server received the event (in UNIX time) and the command. Repeat this process a few times and try changing some of the Perl script's input parameters or commands. Once you are comfortable with the program's operations, click the Stop icon to exit the server.

---

[6] Richard Stevens wrote a series of books on UNIX programming topics, specifically networking issues, which are considered the bible for UNIX programmers.

Now, let's use MallocDebug to debug the program. Click on the Targets tab and select the BuggyServer target. Notice that the MallocDebug library is already part of the project. You can add the library to a project either at compile time or at runtime. At compile time, you add it as a statically linked library as follows:

1. Select Projects→Add Frameworks.
2. Enter **`/usr/lib/libMallocDebug.a`** into the Go text field.
3. Click the Add button.

At runtime, you add it as a dynamic linked library:

1. Select the Executables tab.
2. Add the following environment variables to the Environment Variables list:

```
DYLD_INSERT_LIBRARIES /usr/lib/libMallocDebug.A.dylib
DYLD_FORCE_FLAT_NAMESPACE 1
```

Now, let's use MallocDebug to find the memory problems in the server. Open /Developers/Applications and launch the MallocDebug program. Next, select File→New Window (Command-N) to display the main work area for the program (see figure 4.7):

Figure 4.7
You use the MallocDebug program's main window to enter options and view results of a debugging session.

- *Executable text area*—Holds the full path to the program you wish to debug.
- *Browse button*—Locates the program (you can also enter the program name by hand).
- *Arguments text field*—Contains any command-line arguments for the program you will debug.
- *Launch button*—Runs the program. Once you run the program, this becomes a Stop button, which you use to exit the program.

The next set of controls enables you to change the view or gather more information on the running program. The *call stack browser* (leftmost pop-up menu) permits you to change how you view the *call stack*, or list of currently called functions. Options include standard, inverted, and flat mode. Imagine a program that calls the following functions, in this order: `main`, `foo`, `bar`, `malloc`. Standard mode displays the call stack from left to right in order of the calls, from `main` to `malloc`. Inverted mode displays the call stack from right to left in order of the calls from `malloc` to `main`. Flat mode lists all calls in one window, ordered by memory allocated (see figure 4.8).

The next pop-up menu, called Display Mode, controls how MallocDebug displays information:

- *All*—Displays call stacks for all allocated memory in the program.
- *New*—Displays calls that have allocated memory from a particular execution time. This option is used in conjunction with the Mark and Update buttons.
- *Leaks/Possible Leaks*—Displays the call stack for possible leaks (for example, leaks that come from stale pointers or midblock deallocations).
- *Defined Leaks*—Displays the call stack for all leaks that occur in the application up to this execution point.

Figure 4.8
MallocDebug enables you to display an application's call stack in three forms: flat (top window), inverted (middle window), or standard (lower window).

- *Trashed*—Displays the call stack for allocated memory that contains illegal writes (buffer overwrites or underwrites, for example).

You use the Mark button to take a snapshot of memory allocations from a specific time (when you click on the Mark button) to the current execution time. The Update button gets all new memory allocations from previous point to now. The rightmost menu permits you to change the display from bytes to counts, showing the number rather than the size of memory allocations.

The next three windows display the contents of the call stack. Clicking on functions in each window displays more information. The bottom window lists specific details for each allocated buffer. Double-clicking on an entry opens the Memory Viewer Panel, where you can interactively search and browse memory.

Let's generate some memory leaks and finding them with MallocDebug:

1. In the main MallocDebug window, click the Browse button, navigate the file system until you find the BuggyServer program (located under the build folder), and click the OK button.
2. Enter the port 4444 into the Arguments text field.
3. Before running the program, make sure no other BuggyServer process is running. Click the Launch button to run the server under MallocDebug.
4. Go back to the shell you were using and enter this command:

```
% perl send.pl 4 1 localhost 4444 leak
pass: 0
sending: leak
received: Sat Jul 13 07:45:27 2002

pass: 1
sending: leak
received: Sat Jul 13 07:45:29 2002

pass: 2
sending: leak
received: Sat Jul 13 07:45:30 2002

pass: 3
sending: leak
received: Sat Jul 13 07:45:31 2002
```

5. Notice that MallocDebug has detected a memory problem, stopped the program, and displayed new information in its main window. Select Inverted from the call stack menu and Leaks from the display menu, and click the Update button.

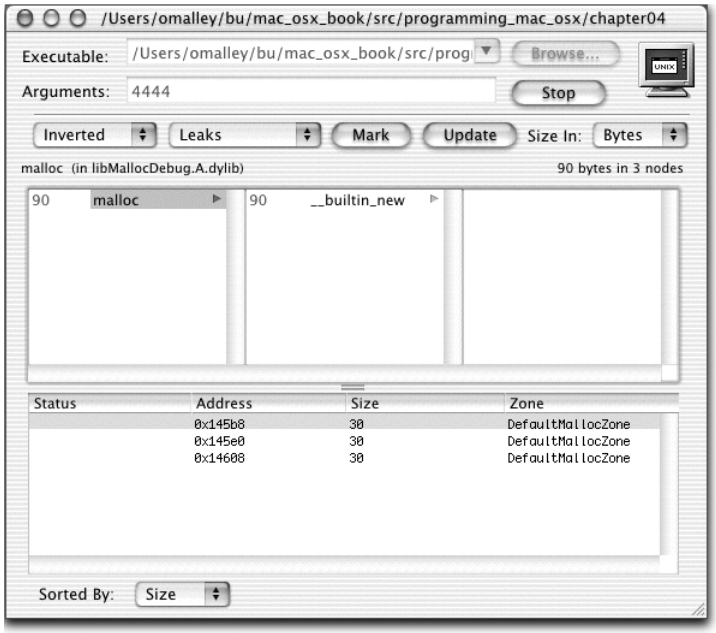

Figure 4.9 The result of the detecting a memory leak in the BuggyServer program

6 Click on the malloc entry in the leftmost window to display the specific memory region in the lower window (see figure 4.9).

7 To get a detailed view of memory, double-click on the first item in the lower window's list. Doing so opens the Memory Viewer Panel and displays more information about the memory layout surrounding the allocation error. As the following hex dump shows, the malloc debug library encloses each allocated memory block with a defined byte sequence (see also the patterns in table 4.1). Remember, the server allocated 30 bytes of memory in the program:

```
0x000145b8: 53617420 4a756c20 31332030 373a3435 Sat Jul 13 07:45
0x000145c8: 3a323720 32303032 0a000000 0000beef :27 2002........
0x000145d8: dead0000 deadbeef 53617420 4a756c20 ........Sat Jul
0x000145e8: 31332030 373a3435 3a323920 32303032 13 07:45:29 2002
0x000145f8: 0a000000 0000beef dead0000 deadbeef ................
0x00014608: 53617420 4a756c20 31332030 373a3435 Sat Jul 13 07:45
0x00014618: 3a333020 32303032 0a000000 0000beef :30 2002........
0x00014628: dead0000 deadbeef 53617420 4a756c20 ........Sat Jul
0x00014638: 31332030 373a3435 3a333120 32303032 13 07:45:31 2002
```

Table 4.1 Patterns the malloc debug library writes to memory

Pattern	# of bytes	Bytes
Sat Jul 13 07:45:27 2002	24	53617420 4a756c20 31332030 373a3435 3a323720 32303032
NULL terminator	1	0a
Empty bytes	5	000000 0000
Defined pattern	20	beef dead0000 deadbeef
Sat Jul 13 07:45:29 2002	24	53617420 4a756c20 31332030 373a3435 3a323920 32303032
NULL terminator	1	0a
Empty bytes	5	000000 0000
Defined pattern	20	beef dead0000 deadbeef
Sat Jul 13 07:45 :30 2002	24	53617420 4a756c20 31332030 373a3435 3a333020 32303032

Let's look at another common type of memory error: illegal memory writes either before or after an allocated buffer. Suppose that through some sort of rogue pointer operation, you happen to write past the end of an allocated buffer or overwrite memory before the allocated buffer. Here's an example of this behavior:

```
char *buf = new char[10];
strcpy(buf, "AAAAAAAAA");
// overwrite past beginning of buffer
strcpy((buf-5), "ZZZZZZZZZZZZZZZZZZZZ");
// overwrite past end of buffer
strcpy((buf+5), "ZZZZZZZZZZZZZZZZZZZZ");
```

These types of bugs are really nasty because they do not always appear when the overwrite occurred; it depends on how memory is laid out. In addition, they can corrupt the core file the program generates when it crashes, making it difficult to track down the initial cause and location of the error. To generate this class of error within the BuggyServer program, enter the following commands:

```
% perl send.pl 1 1 localhost 4444 over-beg
% perl send.pl 1 1 localhost 4444 over-end
```

The first command generates an error by overwriting past the beginning of the buffer; the second command overwrites past the end of the buffer. Generate an

overwrite past beginning of buffer (the first command), select Inverted and Trashed from the menus, and select the last memory record (size 10) from the list. The following example shows a hex dump from MallocDebug showing memory from a buffer overflow:

```
0x000145a8: 00000000 00000001 beefde5a 5a5a5a5a ...........ZZZZZ
0x000145b8: 5a5a5a5a 5a004141 4100beef dead0000 ZZZZZ.AAA.......
```

As you can see, the buffer that should read `AAAAAAAAA` was overwritten and now contains `ZZZZZ ZZZZZ.AAA`. Once again, MallocDebug makes finding and correcting memory-related errors quick and painless.

The MallocDebug program contains many more features than described here. You can use it to debug command-line application as well. See the program's online help for more information.

4.5.9 ObjectAlloc

The ObjectAlloc program provides another dimension to investigating memory-related errors in programs. Like MallocDebug, it enables you to collect and view memory allocations from a target program. Unlike MallocDebug, it does not require you to link your application to any libraries. One of the main reasons to use ObjectAlloc is its playback feature. This feature permits you to single-step forward and backward through all memory allocations your program makes. This technique is particularly useful for applications that contain memory errors in complex data structures that change as the programs executes. Being able to play back all memory-related operations is a real help in diagnosing potential memory-related problems.

4.5.10 PEF Viewer

Carbon applications that run under Mac OS X and Classic mode (Mac OS)[7] store information in a format called Code Fragment Manager (CFM). Within this format are storage areas called PEF containers, which hold PEF information. The PEF Viewer utility lets you graphically view aspects of a PEF container. You can view all the imported/exported symbols, disassemble any code section, disassemble the relocation opcodes, and view compressed and uncompressed data sections.

[7] They can also be mach-o format.

4.5.11 PackageMaker

You can deliver Mac OS X applications in a variety of formats including tar files, GNU zip files, and package formats. You use UNIX formats (tar, gzip) for distributing UNIX command-line programs. You use the package format to deliver Mac OS X programs. PackageMaker creates application distributions for your Mac OS X programs. Basically, you specify the files that make up the package, select some installation options, and tell PackageMaker to create the application package.

4.5.12 Pixie

The Mac OS X User Interface Guidelines define, among other things, the correct layout of user interface components. These layouts can be very exact. For example, the layout of a standard pop-up menu should be 20 pixels high and contain 8 pixels from the text label's trailing colon to the left edge of the menu. In addition, there should be at least 12 pixels between two vertically stacked standard pop-up menus. To ensure your interface is correct, it's helpful to check its interface components by viewing its layout at varying resolutions, from normal view down to the pixel level.

Pixie is a program that lets you view, copy, and save anything on the screen at varying magnification levels (see figure 4.10). With Pixie, you can check the layout of interface items at the pixel level, copy the current image to the clipboard or to a .tiff file, and perform other useful tasks. Another nice feature is Pixie's ability to display the selected pixel's RGB values, enabling you to get exact color values for various interface components.

4.5.13 Project Builder

Project Builder is an IDE that contains an edit, build, and run environment for developing Mac OS X applications. With Project Builder, you can build all types of Mac OS X application, including Carbon and Cocoa applications, bundles and frameworks, kernel extensions, Java applications and applets, plug-ins, and tools. Project Builder uses many of the underlying UNIX tools to perform its development tasks, such as `gcc`, `g++`, `gdb`, and CVS.

4.5.14 PropertyListEditor

You configure most UNIX system services (such as `cron`) and applications such as Apache through some sort of text-based configuration file. Typically, you open the file in your favorite editor, change configuration parameters, and restart the process for the new parameters to take effect. This method of specifying application

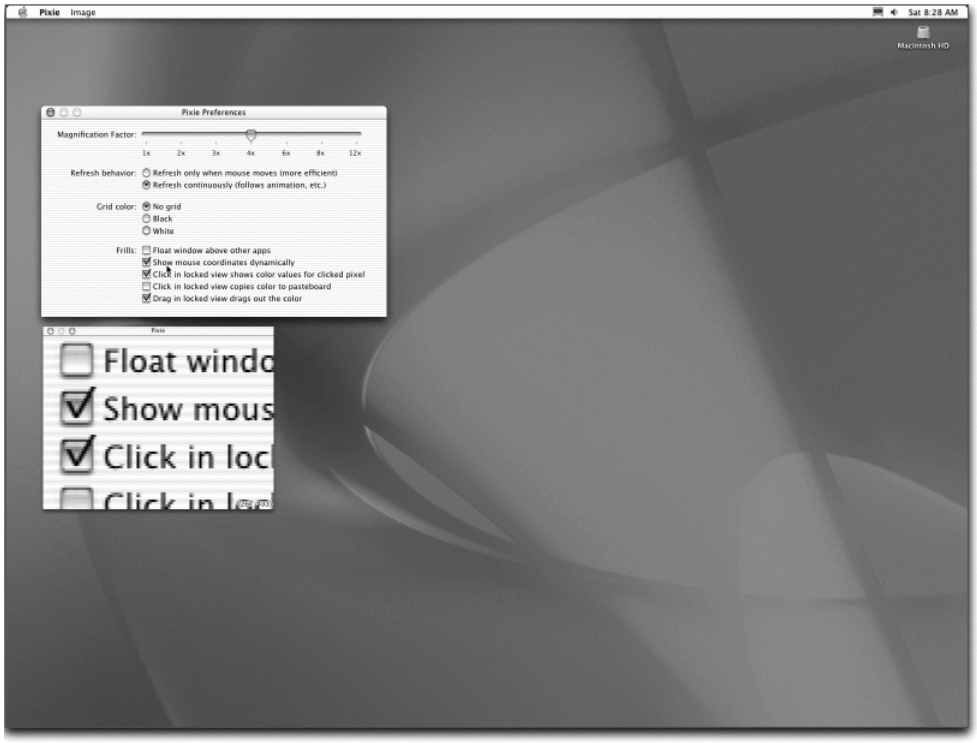

Figure 4.10 Pixie displays regions of the screen at various magnification levels and is useful for checking your program's interface layout.

parameters applies equally to user programs. UNIX systems are full of text-based configuration files, many of which are stored in different formats. Text-based files make it straightforward to reconfigure your system, but you must learn each new configuration file format and make sure you do not make a mistake in editing the file.

Mac OS X builds on this functionality by adding a new configuration file type called a *property list* that holds information in a single, structured format. Property lists are text files that hold information formatted in XML, the near-universal language for storing and exchanging structured data among systems.

Let's look at an example of how Mac OS X stores application parameters using property lists. Open ~/Library/Preferences, find a file with a .plist extension, double-click on the file to launch PropertyListEditor, which will load the file. Figure 4.11 shows the PropertyListEditor displaying the property list file for Adobe Acrobat Reader.

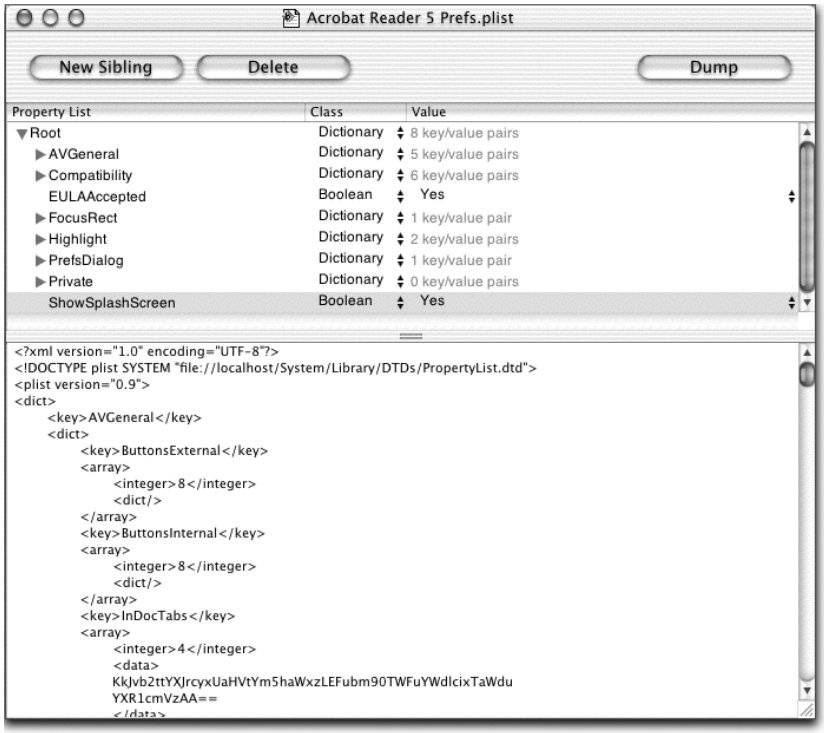

Figure 4.11 An example PropertyListEditor displaying preference information for Adobe Acrobat Reader

Using this interface, you can change the behavior of the application. For example, changing ShowSplashScreen from Yes to No removes the startup splash screen. You can also display (read-only) the file in XML format by clicking on the Dump button.

4.5.15 Quartz Debug

Imagine you are writing a game under Mac OS X and you need to compare several competing algorithms that update the contents of a window. You also need to get detailed information about the memory size of a window buffer. For both tasks, the Quartz Debug program will provide you with the answers. Quartz Debug gives you detailed information about the Quartz graphics system.

When you run the program, it displays a window showing three options that alter the behavior of the program (see figure 4.12).[8]

[8] A new option is added to the application under Jaguar called Flash Identical Updates.

Figure 4.12
You use the QuartzDebug program to get information about Quartz-level operation, such as screen updates and window proprieties.

Enabling the Autoflush Drawing option clears the CoreGraphics graphic context after every drawing action. The Flash Screen Updates and No Delay After Flash options work in tandem, enabling you to see what part of the screen Quartz updates. The Flash Screen option controls whether the screen is marked before the system updates a region of the screen; the No Delay option prevents a delay after the screen is marked.

For example, select the Flash Screen option and point and click various parts of the user interface; move the mouse over a window's close, minimize, and zoom buttons; move a window around the screen; or select an item from an application menu. You should see the following sequence: a yellow region is displayed, showing which part of the screen the system is about to update; a slight delay occurs if the No Delay option is not checked; and then the real screen update takes place.

Clicking the Show Window List button opens a window that lists all windows open on the system (see figure 4.13). Each line provides information about the window, including its connection ID (CID), which tells you what process owns the window; the memory held by the window, in KB; the name of the program to which the window belongs; and the current relative location of the window in pixels (Rect).

4.5.16 Sampler

As software developers, we are always looking for ways to speed up the runtime execution of our programs. Sometimes, in our quest for improvements, we blindly fall into traps that we know are wrong—we ignore them because we are focused too hard on one goal. Every year and a half, Moore's law is realized as hardware gets faster; some developers believe this speed increase eliminates the need to spend time on things like optimization. After all, why waste development time and money when you can throw hardware at the problem, which often costs less than programmer time? This approach works in some cases, but for other classes

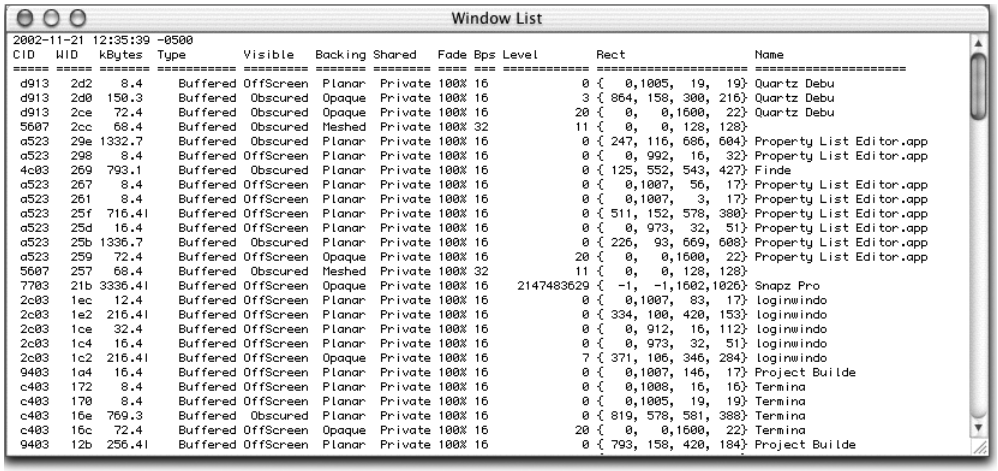

Figure 4.13 The Window List window provides a great deal of information about the windows that are currently open on the system.

of applications it does not help. For example, game-theory simulations and web cache simulators routinely take days or weeks to return results. In these cases, looking for ways to improve program performance can mean the difference between the program returning useful results in days rather than weeks.

Other problems are more concrete and involve basic aspects of a program: for example, how I/O affects program performance, or what delays the program encounters in reading bytes from a socket. As you know, the first rule of optimization is understanding what to optimize before jumping in. It makes little sense to optimize a section of code that will not help the runtime performance of the program. Sampler, a performance and memory analysis program from Apple, is a good choice for performing runtime performance analysis of your program.

Sampler collects four types of performance statistics:

- Samples of call stacks
- Allocation information
- File operations
- Information about specific function calls

Sampler works by collecting discrete samples of the call stack at millisecond intervals. The call stack holds a list of calls the program makes at this point and time. This type of performance analysis tells you the frequency at which the program calls a function, based on the sample rate. Think of it this way: the current execution

state of your program is captured by its current call stack, which, over the course of the program's runtime, is arranged as a linear time sequence. On average, if the sample rate is high enough, you can capture a representation of the program's runtime call history. At each time step, Sampler grabs the current stack frame and increments the count of the frame. When you stop sampling, you have a group of captured stack frames and the number of times they occurred. From this information, you can get a good idea how often your program called a function or method, as well as its context.

Figure 4.14 shows an example of Sampler's main window after sampling the BuggyServer program.

Sampler also lets you capture allocation information so you can track a program's memory allocation behavior. It is similar to MallocDebug, but does not require you to link to any library.

Sampler can also track file-related calls in your application. This feature lets you graphically see all file-related operations in your application and determine what functions or methods are performing file I/O operations. This technique can be useful for many situations. For example, imagine your application's performance is slower than you would like. With this feature, you can examine all

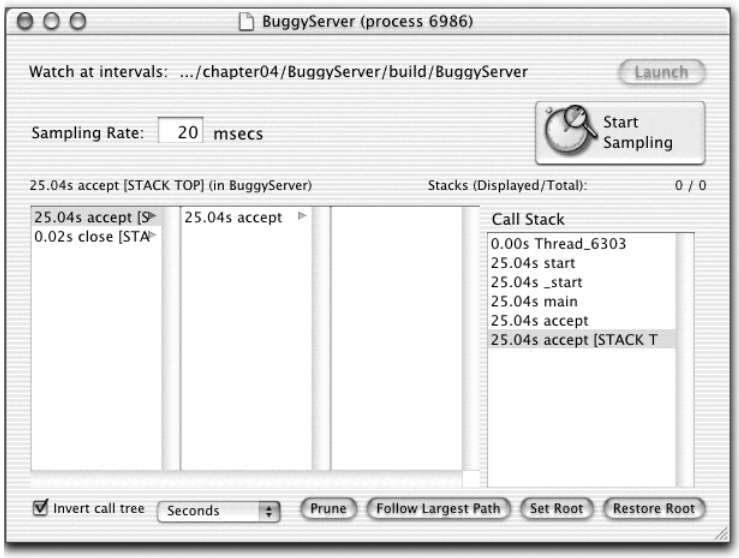

Figure 4.14 Sampler collects runtime statistics for your program and displays the results in the main window. This information is useful in analyzing your program's performance and determining where you need to optimize.

file-related calls and see if the program is calling file operations more frequently than you thought. In addition, you can attach to other programs running on the system or launch them under Sampler, and examine their I/O operations. Doing so is very useful in determining the I/O requirements for other programs you may wish to use on a project, such as database applications like MySQL.

You can get detailed information about specific function calls using Sampler. For example, imagine you need to know how often your program calls a specific function and who is calling the function. You enter the function into the Add Functions Watch list and launch the program. Now, Sampler will collect statistics for this function only and display them when you click Update All Calls.

Overall, Sampler is a useful program with specific features that help you debug or optimize your program. If you have used other CPU-based profiling tools like `gprof`, you may find Sampler's profiling information somewhat limited. However, used in conjunction with other tools, or as a tool that provides a high-level view of a program's runtime performance, it is quite useful. Later in the chapter, you will use a command-line profiling tool to look into the performance of another server program.

4.5.17 Thread Viewer

Normally we view computer programs as performing a series of defined, sequential instructions with the goal of fulfilling some task. Conceptually this is correct, but in some contexts, it is possible to perform instructions in parallel (the program does more than one thing at a time). Imagine a text editor that performs a batch search-and-replace operation on a large file while still enabling the user to open other files. Programs that perform several operations in parallel are called *multithreaded* programs.

In general, threading is wonderful—without it, many problems would be difficult to solve efficiently. But even with all the power and performance benefits of multithreading, it can be difficult to get the implementation right; ask anyone who has worked on a nontrivial multithreaded application. The Apple Mac OS X developer tools come with a program called Thread Viewer that can help you understand how the threads in your program are interacting at runtime and can save you significant development time. Before I describe the program, let's briefly review the relationship between threads and processes. If you are comfortable with this topic, feel free to skip this section.

Threads and processes

A UNIX process runs within a defined address space, has its own internal data structures and resources, and executes a set of instructions until termination. The term *thread* defines a basic unit of execution through a process. In the traditional model, where instructions proceed sequentially, we say that the process has a single thread of execution: if statement B follows statement A, B must wait for A to finish before executing.

For many programs, this is a perfectly acceptable design, and it is simple to implement and understand. However, some programs do not fit into this model and suffer unnecessary performance bottlenecks when designed this way. For example, programs that perform a long series of computations whose subsequent steps do not rely on any preceding step, or servers that must service many connections simultaneously, would benefit greatly from performing instructions in parallel, as apposed to sequentially.

One common technique to address concurrency within the context of single-threaded programs is the `fork/exec` family of calls. These calls enable copies of your program to run as child processes under a common parent. The usual examples of this technique are servers that handle more than one client connection at a time. The servers' goal is to service client requests concurrently. In this context, the server process is the *parent process*. When the server accepts a connection from a client, it creates a copy of itself, called the *child process*, to service the request, and goes back to accepting client connections. It repeats this loop for each client connection until termination.

The child process is in most respects a duplicate version of the parent. The child process independently services the client request and exits while the parent is accepting more connections and forking off more children. The main limitations of this approach are the time and memory required to duplicate aspects of the parent process and the limited information a parent and child process can share. One solution to these performance issues is to implement the server as a threaded server that pools client connections (more on this later).

To write multithreaded applications, you use a set of instructions from a thread library and link the thread library to your program at compile time. A common thread library is pthreads, also called POSIX threads library. The pthreads library implements a set of threading primitives that applications use to add threading to the program. As you can imagine, one of the challenges of writing threaded code is keeping one thread from changing the data another thread needs before the second thread gets a chance to see it. Because threads can share data, you must ensure that you protect all shared data through a locking scheme (such as semaphores or mutexes).

As you know, Darwin (and its Mach kernel) is the core operating system of Mac OS X. Mach kernels use and implement processes and threads differently than most other UNIX implementations. In many UNIX monolithic kernels, the basic level of scheduling is the process, where all threads within the process are bound by the scheduling priority of the process; when a process is suspended by the operating system, all its threads are also suspended. Under Mach, the thread is the basic execution unit, not the process. Thus, under Mach, scheduling priority is handled on a per-thread basis: the operating system coordinates and schedules threads from many different tasks.

Using Thread Viewer

The Thread Viewer program displays a graphical representation of the interaction of your application's threads, as well as providing information on a per-thread basis. When you run the Thread Viewer program and attach to a running process, you have full access to the program's current thread state. You can watch each thread and see if it is running, is blocked on a semaphore or lock, is suspended, or has terminated. This information can be a real time saver.

One of the most frustrating elements of debugging multithreaded applications is asynchrony. Multithreaded programs are inherently asynchronous, and consequently are very difficult to debug because you simply cannot reproduce the chain of events that led to the bug. A tool that graphically displays the state of the threads (at runtime) can help determine the cause of errors. Let's look at an example of how to use Thread Viewer on a threaded server that pools client connections.

The project implements a simple thread server that pools connection threads. When the server starts up, it creates a pool of connection threads. As clients connect, the next available connection in the thread pool handles the request.

To see the server's thread activity, you launch Project Builder, open the project ThreadedServer, and build and run the program. To simulate concurrent client requests, you'll wrap the Perl send.pl script (discussed in section 4.5.8) with another script that calls it several times. Doing so enables you to simulate many clients concurrently connecting to the server:

```
#!/usr/bin/perl -w
use strict;

my ($max) = @ARGV;

for(my $i=0; $i<$max; $i++) {
  system("perl send.pl 1 1 localhost 4444 over-beg &");
}
```

Before running the client script, follow these steps:

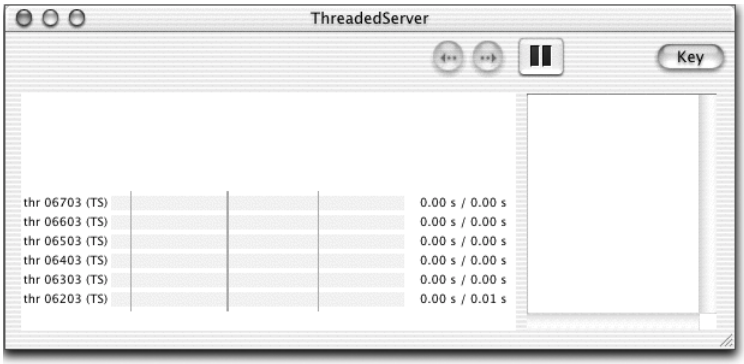

Figure 4.15
An example of the main Thread Viewer window after attaching to the threaded server

1. Launch Thread Viewer (located in /Developer/Applications), select File→Attach, select the threaded server from the application list, and click OK. You will see a window (figure 4.15) that displays the current server threads (five connection threads plus the main thread).
2. Click the Key button to display a drawer that holds a legend of the thread states.

Thread Viewer displays thread activity with a horizontal bar, one per thread, with a tick mark spaced at every time step. You can update the rate at which the application samples thread activity by changing the sampling interval from the Preference menu. To the left of the thread bars are the thread addresses, which uniquely identify each thread. The number to the right of the thread shows the cumulative CPU time consumed by the thread; a high value in relation to the other threads indicates that a thread is consuming excessive CPU time. In general, you would like these values to be as balanced as possible.

Now, continue as follows:

1. Open a shell and run the client script (listed earlier) with the following command:

   ```
   % perl r.pl 100
   ```

   This command sends 100 concurrent messages to the server. While the clients are sending messages, look at the thread timelines (see figure 4.16).
2. You will see flashes of green, indicating that a thread is running, and yellow, showing that it has recently run. Once the script ends, run it a few more times. Ideally, the thread times should be evenly distributed, indicating that no thread is monopolizing the CPU. In addition, click on a thread timeline while the client is sending messages to see the call stack at that interval (see figure 4.17).

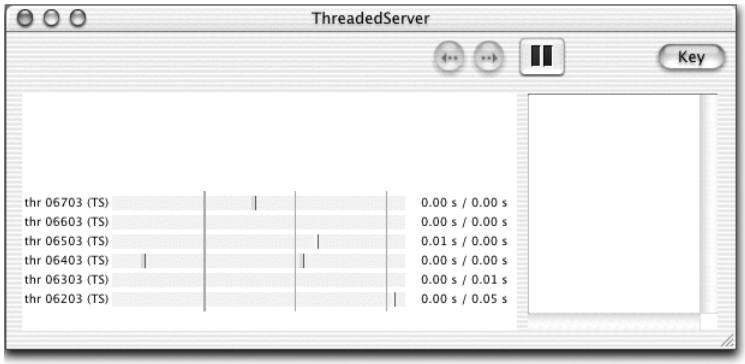

Figure 4.16
The Thread Viewer shows the activity of an application's threads. This example shows the ThreadedServer accepting concurrent client messages.

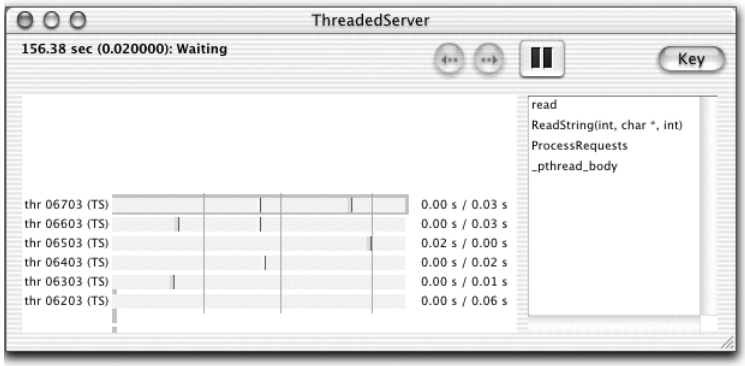

Figure 4.17
By clicking on a thread bar, you can get detailed information on the state of the thread

3. Let's introduce a bug into the server and see how Thread Viewer handles the problem. Stop the server (clicking on the stop sign within Project Builder), open the main.cpp file, and locate the `ProcessRequests` function. Look for a comment within this function and do what it says.
4. Rerun the server and the client script and reattach to the server process within Thread Viewer. Notice how a few threads' timelines change to light pink, indicating that the thread is waiting in a lock. Also notice that the client messages stop (in Project Builder's output window), indicating that the server is blocked. Thread Viewer has immediately alerted you to the fact that there is a problem in your code, and it tells you the type of problem.

As this example demonstrates, Thread Viewer is a good tool for graphically displaying the status of the threads within your program; it lets you quickly locate and fix threading errors.

4.5.18 icns Browser

Traditionally, Macintosh applications stored their application icons within the application file in the resource fork. Under Mac OS X, this arrangement has changed. Most Mac OS X applications are stored as *bundles*. A bundle is a directory that holds programs components in one location, including the application, application resources, and application icons. If you open a shell and change to the directory holding an application, you can easily see this arrangement. You can also view the contents of the folder by holding the Control key, single clicking on the program's icon, and selecting Show Package Contents from the contextual menu.

The Resources directory (located under the application's parent directory) holds the application resource files, including icon files stored in .icns files. You use the icns Browser program to display the contents of a .icns file. It shows the icons in the file for different bit levels and icon masks for the different bit levels (see figure 4.18).

This program is not an editor, but rather a viewer. To create application icons, use the Icon Composer program.

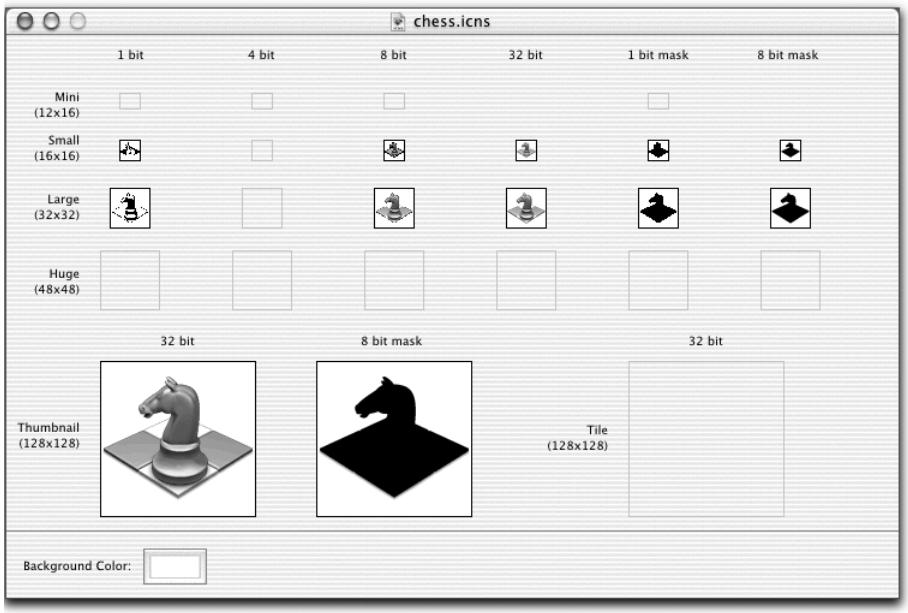

Figure 4.18 The icns Browser program displays application icons stored in its .icns file.

4.6 Apple's command-line development tools

In addition to the GUI-based development tools, Apple has included some very powerful and useful command-line tools for debugging and monitoring Mac OS X applications. You may wonder why you need to use UNIX-like command-line tools for developing Mac OS X GUI applications when GUI tools are available. Mac OS X applications primarily use the Cocoa and Carbon frameworks for their services; but these services use the underlying Darwin operating system, which is a preemptive multitasking system that supports many programs running concurrently. Understanding this interaction and being able to use it to your best advantage can make all the difference between a snappy, properly performing program and a sluggish program that is no fun to use. Currently, the command-line tools supplied by Apple let you peek into the operating system while your program is running and see how it is using the system's resources. This is a powerful tool ability that will help you understand how to design and potentially optimize your program to make the best use of Darwin's power. In addition, the command-line tools offer a greater level of detail than the GUI tools.

Another application of these tools is troubleshooting programs you did not write, but suspect are causing problems (reverse engineering programs). Imagine you are using a script to insert data records into a database. Some insert operations are very slow and cause excessive disk thrashing in the database program. By using the command-line tools, you can get a snapshot of how the database program interacts with the operating system, which may shed light on the cause of the problem.

All the command-line tools are simple to use, but they do require some study to understand their use and features. In truth, you must understand the operating system and the memory allocation scheme, and you need some experience using the tools on real problems. Luckily, man pages are available for all the tools. Some of these tools, like `top` and `gprof`, will be familiar to most UNIX developers; others are specific to the Mac OS X environment. In this section, I will try to minimize repeating information from the man pages and instead concentrate on showing examples of how you can use these tools for common development activities.

4.6.1 ps (process status) and top (system usage statistics)

Both the `ps` and `top` commands will be familiar to most UNIX users. You use the `ps` command to get status information for a process. Its typical command-line invocation is in one of the following forms:

```
Processes:  48 total, 2 running, 46 sleeping... 162 threads            12:10:34
Load Avg:  1.82, 0.94, 0.69     CPU usage:  57.1% user, 42.9% sys, 0.0% idle
SharedLibs: num =  153, resident = 48.2M code, 4.04M data, 15.4M LinkEdit
MemRegions: num = 7485, resident =  203M + 5.74M private,  126M shared
PhysMem:  62.7M wired,  320M active,  248M inactive,  630M used, 9.68M free
VM: 3.26G + 68.6M   112108(112108) pageins, 102748(102748) pageouts

  PID COMMAND      %CPU   TIME    #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
12110 top          0.0%  0:00.21   1    14    14   252K   372K   488K  1.40M
12054 ssh          0.0%  0:00.24   1    13    14   204K   540K   620K  1.48M
12036 tcsh         0.0%  0:00.24   1    16    16   488K   700K   960K  5.74M
```

Figure 4.19 Output of the `top` command on a Mac OS X machine

```
ps aux
ps aux | grep [process-name]
```

The first syntax lists extended information for all process on the system for all users. The second displays the same information, but only for the specified process name.

The `top` command iteratively shows system usage statistics for the top processe. The Mac OS X implementation is somewhat different from those running on other flavors of UNIX. It displays more information that is specific to Mac OS X and gives you a quick snapshot of what is going on in the system. Figure 4.19 shows the output of the `top` command for a Mac OS X machine (Darwin Kernel Version 5.5: Thu May 30 14:51:26 PDT 2002; root:xnu/xnu-201.42.3.obj~1/RELEASE_PPC Power Macintosh powerpc).

Figure 4.20 shows the output of the `top` command for a Linux machine (Linux 2.4.7-10smp #1 SMP Thu Sep 6 17:09:31 EDT 2001 i686 unknown).

```
12:08pm  up 61 days,  2:37, 24 users,  load average: 3.11, 3.29, 3.36
226 processes: 220 sleeping, 4 running, 2 zombie, 0 stopped
CPU0 states: 100.0% user,  0.0% system, 100.0% nice,  0.0% idle
CPU1 states: 99.0% user,  0.0% system, 99.0% nice,  0.1% idle
CPU2 states: 82.0% user,  6.0% system, 78.0% nice, 11.0% idle
CPU3 states: 99.0% user,  0.0% system, 99.0% nice,  0.1% idle
Mem:  2575148K av, 2324376K used,  250772K free,      72K shrd,  147704K buff
Swap: 4708312K av,    3332K used, 4704980K free                1902236K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 9479 omalley    9   0  1088 1084   760 R     9.7  0.0   0:00 top
    1 root       8   0   520  520   452 S     0.0  0.0  20:18 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
```

Figure 4.20 Output of the `top` command for a Linux machine

```
load averages:  0.05,  0.02,  0.02                                      12:11:55
166 processes: 165 sleeping, 1 on cpu
CPU states:     % idle,     % user,     % kernel,     % iowait,     % swap
Memory: 1024M real, 16M free, 377M swap in use, 1380M swap free

  PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
24404 root       1  58    0 2584K 2056K sleep   0:00  0.80% sshd
  136 root       5  58    0 2848K 1888K sleep  17:18  0.53% automountd
24424 omalley    1   0    0 2264K 1952K sleep   0:00  0.48% tcsh
```

Figure 4.21 Output of the `top` command for a Solaris 2.7 machine

The output of the `top` command shown in figure 4.21 comes from a Solaris 2.7 machine (SunOS 5.7 Generic sun4u sparc SUNW,Ultra-60).

As you can see, the Mac OS X implementation provides information about thread usage at both the system and individual process level. Another valuable feature of the Mac OS X version is that if a process's `VSIZE` (the total address space currently allocated) is increasing, the command places a + after the value. This is a quick indicator that the program's memory usage is increasing, which could indicate a memory leak. See the man pages for more detailed information about the `ps` and `top` commands' usage and options under Mac OS X.

4.6.2 sc_usage: showing system call usage statistics

Suppose you are developing a simulation program that requires the processing of large amounts of data. Ideally, you would like to read the data into physical memory, perform your calculations on the data, and write the result. Alternatively, perhaps you are writing a multithreaded program and you need to get detailed information about what system calls the program makes, as well as thread performance, cache hits, and timing. You need a tool that enables you to peek into a program as it runs and view its runtime state. In either case, the `sc_usage` command is a good choice.

The `sc_usage` command samples an application at a specified interval, showing the system calls it makes as well as other information such as the number of generated page faults. This information helps you understand the kinds of system calls your program makes, and also lets you determine potential performance bottlenecks.

Let's use `sc_usage` to look at the threaded server program described in section 4.5.17:

1. Run the ThreadedServer program and get its process identifier (using `top` or `ps`).
2. Open a shell in the Terminal application and enter the following command (you must be root or have root privileges to run this command):

```
% sudo sc_usage [pid-of-server]
```

3. By default, `sc_usage` samples the server application every second. Send it some messages with the client Perl script (see section 4.5.17 for more information). Figure 4.22 shows the output of the program.

As you can see, the output includes a lot of useful information. The upper part of the display tells you the number of threads in the program, the current system time (21:15:31), how long the `sc_usage` command has been running, and some global state information. The next columns of data show the system calls made thus far, the number of times each call was made (from when the `sc_usage` command was run), the CPU time consumed by the command at the current sample time, and the time the process has been waiting. Below this, the output lists the current system calls, the last path name that was blocked, the cumulative thread block time, the thread number, and the thread priority.

```
ThreadedServer    8 preemptions   32 context switches    6 threads    21:15:31
                  1 fault         32 system calls                      0:01:22

TYPE                            NUMBER        CPU_TIME   WAIT_TIME
-------------------------------------------------------------------------------
System         Idle                                      0:14.653( 0:00.076)
System         Busy                                      1:02.225( 0:01.026)
ThreadedServer Usermode                      0:01.131

cache_hit                        84(1)       0:00.002

semaphore_wait_signal_t          54(4)       0:00.001   2:43.608( 0:00.875) W
read                            366(5)       0:00.012   1:30.418( 0:02.361) 4
accept                          185(4)       0:00.006   0:50.727( 0:00.077) W
write                           370(8)       0:00.009   0:00.014
close                           181(1)       0:00.011   0:00.014( 0:00.000)
setsockopt                      185(4)       0:00.001
semaphore_signal_thread          54(4)       0:00.000

CURRENT_TYPE              LAST_PATHNAME_WAITED_FOR     CUR_WAIT_TIME THRD# PRI
-------------------------------------------------------------------------------
accept                                                  0:00.076   0  31
semaphore_wait_signal_t                                 0:00.875   1  31
read                                                    0:00.963   2  31
read                                                    0:00.902   3  31
read                                                    0:00.417   4  31
read                                                    0:00.075   5  31
```

Figure 4.22 Output of the threaded server program using the `sc_usage` command

With this information, you can see that the server makes a lot of calls to the `read` and `write` functions, as you would expect, and a high number of calls to `accept`. You may be able to improve performance by changing the server from an accept-based server to a select server. The information at the bottom of the display is also helpful: it tells you that four of the threads are in the read system call, one is blocked on a semaphore, and the other is waiting on the accept call. It also provides cumulative timing information for each thread.

The output of `sc_usage` provides detailed information about the current state of the program. Compare this with the output of the Thread Viewer program run on the same example—Thread Viewer provides a nice graphical view of the program and threads, but it does not offer the same level of information.

The `sc_usage` command works for a range of applications and types of problems. Try it with programs you did not write, to look at the system call distribution and timing information. It is an excellent reverse-engineering tool, and it is especially useful for looking at Mac OS X programs that you suspect are really Cocoa interfaces that call UNIX commands for services.

4.6.3 fs_usage: reporting system calls and page faults related to the filesystem in real-time

One of the great things about UNIX is the number of cool programming tools that come with the system. Apple continues this tradition by providing a useful file system utility called `fs_usage`. This command presents a continuous display of system-call usage information for file system operations. In its normal form (run with no command-line arguments), it displays information about all instantiated processes except the running `fs_usage` process, Terminal, `telnetd`, `sshd`, `rlogind`, `tcsh`, `csh` and `sh`. A less noisy way to use the command is to supply a process identifier (pid) as its only command-line argument. In this form, it reports all activity for the specified process.

The following listing shows an example of the program's output while monitoring the ThreadedServer program:

```
% sudo fs_usage 3008
11:12:03  read      0.002226 W ThreadedServ
11:12:03  write     0.000017   ThreadedServ
11:12:03  write     0.000070   ThreadedServ
11:12:04  read      1.007925 W ThreadedServ
11:12:04  close     0.000105   ThreadedServ
11:12:04  read      0.000016   ThreadedServ
11:12:04  write     0.000032   ThreadedServ
11:12:04  write     0.000064   ThreadedServ
11:12:06  read      1.044468 W ThreadedServ
```

```
11:12:06  close     0.000066   ThreadedServ
11:12:06  read      0.000014   ThreadedServ
11:12:06  write     0.000021   ThreadedServ
11:12:06  write     0.000063   ThreadedServ
11:12:07  read     1.041570 W ThreadedServ
```

4.6.4 gprof: displaying execution profile data

As you saw earlier, the Apple developer tools come with a program called Sampler that helps you profile your program for performance bottlenecks. Because it uses a sample-based monitor technique, it will tell you the number of times the program called a function, but not the percentage of time each function takes within the program's total runtime. For this type of analysis, `gprof` is the right tool. GNU `gprof` displays the execution profile of a program. Let's look at an example of how you can get this type of information using `gprof`.

The SamplerServer project implements two versions of a server, each with a different socket read function. The first, called SlowServer, reads from a socket one byte at a time until it reaches the string terminator. The second, called FastServer, reads a specific number of bytes from the socket. Richard Stevens points out this problem and discusses design choices and solutions in his classic book on network programming; as Stevens points out, the byte-by-byte version spends most of its time in the kernel (trapping the kernel with repeated system calls), whereas the other version significantly reduces kernel noise and improves performance. Both server implementations enable you to test this behavior and see for yourself the performance differences.

The project is set up with two targets: one for the slow socket read (`SlowServer`) and one for the fast read (`FastServer`). Let's test them, use `gprof`, and evaluate the results:

1. Select the Targets tab, select `SlowServer` from the Target list, select the Build Settings tab in the Editor pane, and make sure the Generate Profiling Code checkbox is selected (under Compiler Options).
2. Build and run the program (Command-R).
3. Open the Terminal application, open a new shell, change to the project directory, and run the send script as follows. Make sure you send the server a very long string (say, longer than 3000) so you can really look at the socket read times. This script sends the string 100 times to the server, pausing one second between sends:

```
% perl send.pl 100 1 localhost 4444 [enter-long-string]
```

4. After reading 100 messages, the server will exit and generate a profiling script called gmon.out. The `gprof` program uses the gmon.out file to print program statistics. To view the program's runtime statistics, change to the build directory and enter the following command:

```
% gprof SlowServer.app/Contents/MacOS/SlowServer gmon.out | less
```

5. Notice the full path to the executable. SlowServer.app is the bundle, not the executable program, so you need to specify the full path to the executable within the bundle.

NOTE The `gprof` tool has been used for years by UNIX programmers to generate execution profiles of programs. Here's how it works:

1. The first step is to add the `–pg` option when building the program you wish to profile. This option tells the compiler to compile source files for profiling and link with the profiling library. For example, to compile and link for profiling, use the following commands:

```
% gcc –o foo foo.c bar.c –g -pg
```

2. Now that the program is built for profiling, you can run it to generate the profiling information or the profiling profile. Run the program as you normally would and let it execute and exit as usual.
3. When the program exits, it writes to the current directory a file called gmon.out. This file contains the program's runtime profile.
4. Run the `gprof` tool, passing it the gmon.out file, which displays the program's runtime execution profile:

```
% gprof gmon.out > gprog.log
```

5. Repeat this process for the `FastServer` to get its performance statistics.

Figure 4.23 shows the output of the `gprof` program for the `SlowServer` implementation (reading from the socket one byte at a time).

The output for the `FastServer` implementation (reading a specific number of bytes from the socket at one time) is shown in figure 4.24.

The program spends 78.9 percent of the total runtime in the system call `read`, accounting for 0.15 seconds. The fast version is quite different; runtime statistics are so negligible that they do not even show up in the profiling output.

From this example, you can see that using `gprof` to profile your program can help you pinpoint and diagnose potential performance problems. In addition, try running the same example with the Sampler program and compare the result.

```
granularity: each sample hit covers 4 byte(s) for 5.26% of 0.19 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 78.9       0.15     0.15                             _read [1]
 15.8       0.18     0.03      100     0.30     0.30  _ReadString__FiPci [2]
  5.3       0.19     0.01                             ___sflush [5]
  0.0       0.19     0.00      100     0.00     0.00  _WriteString__FiPci [22]
  0.0       0.19     0.00        1     0.00    30.00  _main [3]
```

Figure 4.23 Output of the `gprof` program for the `SlowServer` implementation

```
granularity: each sample hit covers 4 byte(s) no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  0.0       0.00     0.00      100     0.00     0.00  _ReadString__FiPci [17]
  0.0       0.00     0.00      100     0.00     0.00  _WriteString__FiPci [18]
  0.0       0.00     0.00        1     0.00     0.00  _main [19]
```

Figure 4.24 Output of the `gprof` program for the `FastServer` implementation

Although doing so is like comparing apples to oranges, it will give you a feel for how these tools differ and how to use them together to solve certain problems. For more information, see `gprof`'s man page, as well as its GNU documentation.

4.6.5 *leaks: searching a process's memory for unreferenced malloc buffers*

The Mac OS X development tools provide several programs you can use to detect memory leaks in your application. A memory leak occurs when you allocate memory within a program and never free it. The command-line tool called `leaks` performs a similar role as the GUI-based profiling tools discussed earlier in the chapter: detecting malloc-allocated memory locations where your program has lost the pointer to the allocated memory, causing a memory leak. The `leaks` command takes one argument, the pid of the process you wish to examine.

Let's look at a simple example. The LeaksExample project implements a simple example of a memory leak. Open this project in Project Builder and build and run the program. You will see several logging messages in the output window followed by the program displaying a window. While the program is still running, get its process ID from the output window and run the following command in a shell (you must be user root or have root privileges to run the `leaks` command):

```
% leaks 11786
Process 11786: 7424 nodes malloced
Process 11786: 7 leaks
Leak: 0x0008fda0  size=46
        0x80813ff0 0x80813ae0 0x80813ffc 0xa1b1c1d3
        0x0008fd00 0x00091ef0 0x00091f70 0x00000000
        0x00000000 0x00000000 0x00000000
Leak: 0x0008fcd0  size=46       instance of 'NSCFDictionary'
        0x00058610 0x00010395 0x00000003 0x00000003
        0x00000004 0xa1b1c1d3 0xa1b1c1d5 0x00000000
        0x0008fda0 0x0008fdb0 0x00000000
Leak: 0x0007f340  size=46
        0x00530068 0x006f0075 0x006c0064 0x0020006e
        0x006f0074 0x00200073 0x00650065 0x0020006d
        0x0065002e 0x00000000 0x00000000
Leak: 0x0007f040  size=46
        0x00530068 0x006f0075 0x006c0064 0x0020006e
        0x006f0074 0x00200073 0x00650065 0x0020006d
        0x0065002e 0x00000000 0x00000000
Leak: 0x0008fcb0  size=30       instance of 'NSUserDefaults'
        0x808190bc 0x00082850 0x0008fcd0 0x0008e980
        0x00000000 0x00000000 0x00000000
Leak: 0x0007f370  size=30       instance of 'NSCFString'
        0x80160880 0x000107f0 0x0007f340 0x00000012
        0x0007f2b0 0x00000000 0x00000000
Leak: 0x0007f320  size=30       instance of 'NSCFString'
        0x80160880 0x000107f0 0x0007f040 0x00000012
        0x0007f2b0 0x00000000 0x00000000
```

As you can see, the `leaks` command detects that the program contains memory leaks and provides you with information about their locations.

Let's look at the format of the information using the last record in the display (italicized in the code). The first line lists the address of the leaked memory block, its size (in bytes), and the source (in this case, an instance of the `NSCFString` class). The next series of lines shows the contents of the allocated memory buffer in hexadecimal. You can use the `-nocontext` option to suppress displaying the allocated memory contents:

```
% leaks -nocontext 11786
Process 11786: 7424 nodes malloced
Process 11786: 7 leaks
Leak: 0x0008fda0  size=46
Leak: 0x0008fcd0  size=46       instance of 'NSCFDictionary'
Leak: 0x0007f340  size=46
Leak: 0x0007f040  size=46
Leak: 0x0008fcb0  size=30       instance of 'NSUserDefaults'
Leak: 0x0007f370  size=30       instance of 'NSCFString'
Leak: 0x0007f320  size=30       instance of 'NSCFString'
```

This information should give you a good idea where your program is leaking.

4.6.6 heap: listing all the malloc-allocated buffers in the process's heap

The `heap` command is a experimental BSD tools that displays memory objects, including Objective-C objects, allocated on the heap of the specified process. You run the command, passing it the pid of the program you wish to monitor. The following listing shows a condensed example of `heap`'s output:

```
% heap [pid]
% sudo heap 3186 | more
Process 3186: 6 zones
Zone CoreGraphicsDefaultZone_0x1671d0: Overall size: 256KB;
278 nodes malloced for 48KB (18% of capacity); largest unused:
[0x001
7331e-207KB]
Zone kCFAllocatorNull_0x701e6944: Overall size: 0KB
Zone kCFAllocatorMalloc_0x701e6914: Overall size: 0KB
Zone DefaultMallocZone_0x11f1d0: Overall size: 852KB;
6849 nodes malloced for 618KB (72% of capacity);
largest unused: [0x01eed88
e-205KB]
Zone Custom CFAllocator_0x701e698c: Overall size: 0KB
Zone kCFAllocatorSystemDefault_0x701e6928: Overall size: 0KB
All zones: 7127 nodes malloced - 666KB

------------------------------------------------------------------
Zone DefaultMallocZone_0x11f1d0: 6849 nodes (632582 bytes)

<not Objective C object>        = 6424 (613064 bytes)
NSMenuItem      = 52 (4056 bytes)
NSDynamicSystemColor    = 29 (870 bytes)
NSImage = 21 (630 bytes)
NSBitmapImageRep        = 20 (1240 bytes)
NSMethodSignature       = 20 (2280 bytes)
NSPathStore2    = 15 (1474 bytes)
NSMenu  = 10 (300 bytes)
NSCarbonMenuImpl        = 10 (140 bytes)
NSCachedWhiteColor      = 9 (126 bytes)
NSDistantObject = 3 (74 bytes)
NSConcreteMutableData   = 3 (90 bytes)
NSWindowGraphicsContext = 3 (74 bytes)
NSBundle        = 2 (92 bytes)
NSView  = 2 (188 bytes)
```

4.6.7 malloc_history: showing malloc allocations that a process has performed

The `malloc_history` command is another command-line tool that detects nonfreed memory allocations and buffer overwrites in your application. To use the command, follow these steps:

1. Open a shell and set the environment variables `MallocStackLogging` and `MallocStackLoggingNoCompact` to value `1` (or place them in your .csrch file):

```
% setenv MallocStackLogging 1
% setenv MallocStackLoggingNoCompact 1
```

2. Make sure you compile the program you wish to debug with debugging turned on (either through the `–g` option or by checking the Generate Debugging Symbols checkbox in Project Builder, under the Target Build settings).
3. Run the program. While it is running, run the following `malloc_history` command. (The leading `clear` command will clear the current shell's output, enabling you to view the results more easily.) If the program contains leaks, `malloc_history` displays them, along with the call stack:

```
% clear; malloc_history 11941 -all_by_size

6 calls for 96 bytes: thread_800013b8 |0x0 | start | _start |
main | malloc | malloc_zone_malloc
```

The `malloc_debug` command has many additional options than are described here. For more information, see the command's man page.

4.6.8 sample: profiling a process during a time interval

The `sample` command acquires performance statistics for the specified application by sampling its execution at an interval specified by the user. It gathers data the same way as the GUI Sampler application, covered in section 4.5.16. The command takes three arguments: the pid of the process to monitor, how long the command should sample the program (in seconds), and the sampling rate (in milliseconds). For example, the following command collects performance statistics for the SampleExample program:

```
% sample 12525 20 10
Sampling 12525 each 10 msecs 2000 times
Now analyzing results...
Samples: 42696 bytes
Analysis written to file /tmp/SamplerExample_12525.sample.txt
```

As you can see, a performance report is written to the /tmp directory, which contains a textual representation of the collected statistics. Here's an example of the output generated by the `sample` program:

```
752 UnOptimized_LoopFusion(double *, double *, int)
  392 sqrt
  361 sqrt [STACK TOP]
```

```
  31 nan
  31 nan [STACK TOP]
273 UnOptimized_LoopFusion(double *, double *, int) [STACK TOP]
  61 error_message
  61 error_message [STACK TOP]
  26 rest_world_eh_r7r8
  26 rest_world_eh_r7r8 [STACK TOP]
359 Optimized_LoopFusion(double *, double *, int)
  237 sqrt
  219 sqrt [STACK TOP]
  18 nan
  18 nan [STACK TOP]
74 Optimized_LoopFusion(double *, double *, int) [STACK TOP]
  31 error_message
  31 error_message [STACK TOP]
  17 rest_world_eh_r7r8
  17 rest_world_eh_r7r8 [STACK TOP]
```

The `sample` command is a quick, simple way to get performance information about a running program, providing information similar to the Sampler program.

4.7 Summary

Mac OS X provides UNIX developers with all the tools they are accustomed to under their favorite UNIX distribution. Throughout this chapter, you have seen the editing environments, static analysis tools, version control systems, and build tools that are available, and some examples of the tools in action. In addition to the standard tools, many developers are implementing Mac OS X–specific versions of many of the tools that take advantage of the Aqua interface. These programs augment—and in some cases replace—the native tools that come with the Mac OS X system. Overall, experienced UNIX developers will feel right at home developing under the Mac OS X command-line environment.

This chapter has also given you an overview of the development tools that Apple provides with the Mac OS X development tools. These, combined with the traditional UNIX development tools, give developers the power to fully understand the performance and behavior of their programs. This is very important, because it can give you insight into making your applications perform and work better with the underlying operating system. This chapter has provided a taste of the tools and their power. To really understand them, you need to dig in and use them repeatedly on real projects. That way, you will develop experience in understanding the interaction between your programs and the Mac OS X operating system.

Part 3

Programming

Now that we know the underpinnings of Mac OS X, it is time to tackle concrete programming examples. The chapters in this part of the book guide you through the steps of building three working programs using the Apple development tools and technologies you have learned about thus far. In this section, you will develop a Cocoa-based program that puts a GUI on the UNIX tool `wget`, and develop two AppleScript programs—one for organizing a music collection in iTunes, and the other for tracking and displaying system resources. In addition, you will learn some of the more interesting aspects of Jaguar, Apple's newest Mac OS X release.

5 Objective-C and the Cocoa development frameworks

- An overview of Objective-C
- Cocoa software infrastructure
- Memory management and Cocoa
- Design Patterns and Cocoa
- Other Cocoa development languages

The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.

—D. E. Knuth, 1967

This chapter begins your journey into Mac OS X programming by introducing you to Cocoa, Apple's object-oriented framework for developing Mac OS X applications. Your first stop will be an introduction to Objective-C and design patterns. (Cocoa supports two main development languages, Objective-C and Java; this book focuses on Objective-C.) Design patterns are used extensively in the Cocoa frameworks and are important elements of Cocoa application development. The Objective-C section is not so much a language tutorial as an overview of the language's features; it's intended to highlight aspects you will encounter when learning Objective-C. After reading this chapter, you will be well on your way to understanding the Cocoa frameworks and knowing how to use them to write your own applications.

5.1 Introduction

Up to this point, I have covered topics pertaining to the Mac OS and Mac OS X system, its design, and the development tools and frameworks. In addition, I've discussed the basics of Apple's Project Builder and Interface Builder. Project Builder is Apple's Integrated Development Environment (IDE), used for developing all types of Mac OS X applications from command-line tools to GUI-based Aqua applications. The Project Builder IDE uses UNIX development tools such as `gcc`, `g++`, `gdb`, RCS, and CVS for performing its development tasks. This strikes a nice balance between the usefulness of a GUI-based development environment and the power of the UNIX tool set. Interface Builder works in conjunction with Project Builder. You use Interface Builder to design and build the user interface component of your program, as well as define many of your application's classes.

In the first four chapters, you've also learned that the foundation of Mac OS X is Darwin, an open source operating system based on a Mach kernel and BSD. The source code for Darwin is freely available for download, study, and modification (http://developer.apple.com/darwin/index.html). On top of Darwin are the Mac OS X-specific layers that complete the system and help distinguish Mac OS X from other consumer operating systems.

Against this backdrop, you are ready to move on to programming under Mac OS X and learn about its supporting tools and frameworks. As you will see, making

the transition to the Mac OS X develop environment is not as difficult as you may think. If you already know the basics of UNIX-style development, making the transition to Mac OS X should be straightforward. In addition, projects like Fink (http://fink.sourceforge.net/index.php) are actively porting many UNIX tools to Mac OS X, thereby filling the gap between the UNIX tools you get with Mac OS X and those available under other UNIX distributions.

When you're developing applications under Mac OS X, you should be mindful to steer toward utilizing the strengths of the system: its strong support for developing programs with modern user interfaces. For example, writing an application's user interface using X Window widgets makes little sense when Mac OS gives you Aqua and the Cocoa frameworks.

5.2 Introduction to Objective-C

You develop programs in Cocoa in either Objective-C or Java. This book uses Objective-C for application development. If you have never programmed in Objective-C, don't worry—the language is straightforward to learn and intuitive once you know the basics. I hope that over the next few years other development languages will be added to the mix, so Cocoa programs can be written in languages such as C++, Perl, Python, and Ruby.

This section provides a high-level overview of the Objective-C language. I am assuming you already program in C and understand the basic concepts of object-oriented programming (OOP). If you need more information about C or OOP, see the "Resources" section at the end of the book. This introduction covers the basics of the language with the goal of providing enough information and context for you to understand the language's features, and read and write basic Objective-C code.

In the 1980s, Brad J. Cox developed Objective-C by adding object-oriented Smalltalk-80 features and extensions to the C language. Objective-C is ANSI C with additional features for defining classes, create instances of objects, and send messages to objects. Stepping back a bit, two categories of object-oriented languages exist: statically and dynamically typed languages. C++ and Simula are examples of statically typed languages; Smalltalk, Perl, Python, and Objective-C illustrate dynamically typed languages.

Statically typed languages require programmers to provide type information for all data types at compile time. One of the advantages of a statically typed language is that you can determine typing errors up stream (at compile time), reducing possible runtime errors. Such a language also makes for potentially safer code because you can apply static analysis tools like `lint` to verify type correctness. Generally, statically

typed languages are more efficient than dynamically typed languages because type information does not need to be resolved during runtime. Examples of statically typed languages include C, Simula 67, Java, and C++. C++ is traditionally associated with the Simula 67 school of OOP because you provide type information at compile time to ensure that objects receive the correct messages.

Dynamically typed languages determine a program's type information at runtime, relieving you from encoding type information when you write the program. The runtime system is responsible for tracking a variable's type at runtime and correctly resolving conversions between data types. This typing scheme permits more flexibility at the expense of runtime performance. Examples of dynamically typed languages include Smalltalk, Perl, Python, and Objective-C.

If you already know C and understand the basics of OOP, learning Objective-C should be easy. You can expect to learn the basic concepts of the language in a few weeks and be able to write basic programs in less than a month. What follows is a breakdown of the main aspects of the Objective-C language, above its ANSI C foundations. Objective-C is based on ANSI C, adding object orientation in the style of Smalltalk (dynamically typing). Thus you can mix ANSI C code in the context of the Objective-C class scheme. In fact, a common use of Objective-C is to write wrapper classes for C code. Using this approach, Objective-C functions as a glue language within a language; it joins Objective-C and C code. This arrangement is similar in spirit to using scripting languages such as Perl and Python to glue together various compiled programs.

5.2.1 Object-oriented terminology

Before I discuss the object-oriented features of Objective-C, let's make sure we are using common terminology with a quick review of some basic object terms. Object-oriented systems let you create user-defined data types, which are called *classes*. A class binds into one functional unit data, called a *data member* or *field*, and a set of operations that act on the data, called *methods*. This process *encapsulates* a class's data and the methods that operate on the data into a single entity. Classes are brought to life as *objects*, which are sometimes called *class instances*. Think of a class as the blueprint and the object as the realization of the blueprint.

When you design an object-based system, you typically create classes that represent the domain you are modeling and use them to stipulate specific behavior. These classes usually mirror various elements of the problem domain. For example, if you were designing a program that models a banjo, you would naturally create a class called `Banjo`. This class encapsulates the general properties and operations common to all banjos. To specify the differences between banjos (4 string tenor, 5

string open back, 5 string resonator, fretless, etc.) you use *inheritance* to customize behavior. In the inheritance relationship, the general class is called the *base*, or *parent* class, and the class that specifies custom behavior is the *derived*, or *child* class. Inheritance enables you to create class hierarchies that derive specific behavior from a common parent or set of parents (*multiple inheritance*).

An alternative to inheritance is *composition*. Whereas inheritance derives general behavior from a parent class, composition enables specification by assembling various classes within another class and calling the composed objects through their class interface.

One of the best discussions of object-oriented concepts is the first chapter of *Design Patterns: Elements of Reusable Object-Oriented Software,* listed in the reference section at the end of the book.

5.2.2 Classes

Like all object-oriented languages, Objective-C supports classes. A *class* is a definition, or blueprint, of a user-defined type. Classes are a fundamental piece of any object-oriented language; they encapsulate data members and the methods that operate on the data members. Classes in Objective-C are implemented in two files: the interface definition resides in the .h file, and the implementation resides in the .m file. The following example shows a skeleton of an interface definition (.h file):

```
@interface Class Name : <Super Class> <Protocol List>
{
  // Instance variables
}
// Methods
@end
```

An Objective-C class definition begins with the `@interface` directive. In Objective-C, the `@` token is a compiler directive. Following the `interface` keyword are the class name and its super class. The class name is the name you give to your class; the super class is optional. If it's included, it specifies the parent class from which your class derives its behavior. If it's omitted, your class is a root or base class. Next is the protocol list. Protocols are discussed in section 5.2.5. This is followed by the class's data members, enclosed in a right and left brace, and any class methods. You terminate the class definition with the `@end` directive.

Class implementations reside in a .m file. Here's a skeleton of a class's implementation:

```
@implementation <Class Name>
  // Implement class methods here…
@end
```

You control access to the class through the `private`, `protected`, and `public` keywords. The `private` keyword means that class members are only accessible from within the class that declared them. `Protected` restricts access to inheriting classes, and `public` permits anyone to access the class.

Data members

Data members (sometimes called *fields*) store the data state of a class and provide runtime data persistence for the class. Objective-C uses the same built-in types as C, including int, long, float, double, char, and pointers. In addition, it defines further types exclusive to Objective-C (see table 5.1).

Table 5.1 Objective-C uses C's data types, but also defines further types.

Type	Description
id	Holds an object (pointer); capable of holding any object type
Class	Class definition
SEL	Selector; an internal identifier for a method name
IMP	Pointer to a method returning an id
BOOL	Boolean data type: YES or NO
nil	Null object pointer
Nil	Null class pointer

Methods

Method is an object-oriented term for what we call a *function* in procedural programming languages. However, in OO languages, we tend to think of methods as receiving messages. In Objective-C, like other OO languages, you refer to a method through a class instance variable or the class itself. In the latter case, the methods are called *static methods*.

The following listing shows some examples of Objective-C methods:

```
// A method with no arguments, returning an Object
- foo;
// A method with no arguments, returning an integer
- (int)foo
// A method with one argument, returning an integer
- (int)foo : (int) n;
// A method with two arguments, returning void
- (void)foo: (int) x and: (int) y;
// A method with three arguments, returning void
- (void)foo3: (int) x and: (int) y and: (int) z;
```

Method declarations in Objective-C are preceded by either a minus sign (-) or a plus sign (+). Preceding a method name with a minus sign indicates that it is an instance method, which you can access only through a class instance. The leading plus sign means the method is a class method (or static method), so you can access it only using the class name.

5.2.3 Messages

Object-oriented systems are typically composed as collections of class instances that communicate with one another through message passing. Generally, this is a useful way to view object-oriented systems and makes for a good conceptual separation from more static, procedural systems implemented in languages such as C.

You format Objective-C messages as follows:

```
[receiver message];
```

Using Smalltalk nomenclature, you send the `message` to object `receiver`. For example, the message `[myrect display]` asks the `myrect` object to respond to the `display` message.

In addition to the basic syntax, you can pass arguments along with the message:

```
[receiver message:arg1:arg2];
```

The following example shows some common message-passing scenarios:

```
@interface MyClass2 {
}

- (void) draw;
- (void) draw:(int) n;
- (void) draw:(int) n:(int) color;
- (void) draw:(int) n:(int) color:(int) shape;
@end

[foo1 draw];
[foo1 draw:1];
[foo1 draw:1 :2];
[foo1 draw:1 :2 :2];
```

You can also specify a description for each parameter. When you first encounter this syntax, it seems a bit verbose. However, as you use it in your code, you'll find that it documents the intent of the message parameters:

```
@interface MyClass : NSObject {
}
- (void) draw;
- (void) draw:(int) n;
- (void) draw:(int) n theColor:(int) color;
```

```
-(void) draw:(int) n theColor:(int) color theOutline:(int) shape;
@end

[foo draw];
[foo draw:1];
[foo draw:1 theColor:2];
[foo draw:1 theColor:2 theOutline:2];
```

`NSObject` is the parent for most of the Objective-C class hierarchies.

5.2.4 Categories

Imagine you have a class called `Beer` that implements the basic attributes and behavior of beer. The properties held by this class apply equally to all types of beer. Rather than re-implement these properties for each kind of beer, you make the `Beer` class a base (or parent) class and derive other beers from it, which absorb the basic attributes and behavior of the parent class. For each new beer, you implement only attributes and behavior specific to that type of beer, and reuse the basic properties from the parent `Beer` class.

In OOP, this process is called *inheritance*. Inheritance enables you to use the attributes (data) and behavior (methods) of other classes as a starting point for creating specialized versions of a class. In doing so, you create hierarchies of objects, each a specialization of its parent(s). Many OO languages support both *single inheritance*, where you inherit from a single class, and *multiple inheritance*, where you inherit from multiple classes. However, Objective-C does not support multiple inheritance.

You can also extend class properties through *categories*. Categories enable you to add new methods to a class without using inheritance. For some applications, categories offer advantages over inheritance and are a good way to enhance an existing class.

You can specify a category in an interface and implementation file. The syntax for categories is as follows:

```
// .h file.
@interface Class Name (Category Name) <Protocol List>
// Category methods.
@end

// .m file.
@implementation Class Name (Category Name)
// Category methods.
@end
```

Here's the `Beer` skeleton class and an example of extending the class through an Objective-C category called `BeerAddition`:

```
// Beer.h
#import <Foundation/Foundation.h>
@interface Beer : NSObject {
  float alcoholContent;
}
- (float) getAlcoholContent;
- (void) setAlcoholContent:(float)n;
@end
```

```
// Beer.m
#import "Beer.h"
@implementation Beer
- (float) getAlcoholContent {
  return alcoholContent;
}
- (void) setAlcoholContent:(float)n {
  alcoholContent = n;
}
@end
```

```
// BeerAdditions.h
#import <Foundation/Foundation.h>
#import "Beer.h"
@interface Beer (BeerAdditions)
- (void) printAlcoholContent;
@end
```

```
// BeerAdditions.m
#import "BeerAdditions.h"
@implementation Beer (BeerAdditions)
- (void)printAlcoholContent {
  // Float compares are not a great idea.
  if (alcoholContent <= 0.0) {
    NSLog(@"no alcohol content: %f\n", alcoholContent);
  }
  else if (alcoholContent <= 1.0) {
    NSLog(@"why not drink water: alcohol content: %f\n", alcoholContent);
  }
  else if (alcoholContent <= 5.0) {
    NSLog(@"getting better: alcohol content: %f\n", alcoholContent);
  }
  else {
    NSLog(@"much better: alcohol content: %f\n", alcoholContent);
  }
}
@end
```

```
// main.m
#import <Foundation/Foundation.h>
#import "BeerAdditions.h"

int main (int argc, const char * argv[]) {
  id myBeer = [[Beer alloc] init];
```

```
    float content = 5.2;

    [myBeer setAlcoholContent:content];
    content = [myBeer getAlcoholContent];
    [myBeer printAlcoholContent];

    [myBeer dealloc];
    return 0;
}
```

5.2.5 Protocols

A protocol, in the normal use of the term, defines a set of rules that, when followed, enable interaction between entities. For example, File Transfer Protocol (FTP) is a set of rules that an FTP client and FTP server implement in order to transfer files.

In Objective-C, protocols enable you to declare a list of methods that are not associated with any class and that have no implementation. Any class in your program is free to supply an implementation for the methods. A class conforms to the protocol by supplying an implementation of the protocol methods. A protocol is similar to, but less restrictive than, a Java interface.

You can use Objective-C protocols to implement multiple inheritance. In this context, protocols let a class use specific functionality from different classes without taking on the baggage of the entire class. In addition, protocols enable type checking of objects by specifying that they conform to the same protocol (see http://www.dekorte.com/Objective-C/Documentation/Language/Protocols.html).

You define a protocol as follows:

```
@protocol ProtocolName
  // methods
@end
```

To find out if an object conforms to a specific protocol, use the `conformsTo` method. The method returns `YES` or `NO`, depending on whether the object conforms:

```
[object conformsTo:@protocol(ProtocolName)];
```

5.2.6 Other features

Objective-C provides some other useful features. For example, it lets you save a class's data to disk and bring the object back to life from disk. This process is called *object persistence*. Object persistence enables you to save the current state of an object to disk and later reload the object back into memory with its data intact. Java also implements a form of object persistence called *serialization*.

Distributed objects are a hot topic these days. This technology is implemented in many forms, including Microsoft .NET, UNIX's Remote Procedure Calls (RPC), and Java's Remote Method Invocation (RMI). The basic idea of all these schemes is to set up a protocol and infrastructure so that objects running within different address spaces, typically on different machines across a network, can communicate and request one another's services.

Distributed objects are implemented in Objective-C as *Portable Distributed Objects* (PDO), which let you send messages to an Objective-C object running within another program, another computer, or a host somewhere on the Internet.

5.2.7 Why learn Objective-C?

Before leaving the topic of Objective-C, I'll address a common concern. Many UNIX developers question the value of learning Objective-C, feeling that it's an Apple-only technology with little application in other areas. In addition, Cocoa also supports Java, a far more popular language, so why invest the time in learning Objective-C?

Let me give you my rationale for using Objective-C for Cocoa development. Apple recommends using Cocoa for developing new applications for Mac OS X; if you are going to write new Mac OS X applications, use Cocoa. At this time, Cocoa supports two languages: Objective-C and Java. Objective-C is not a mainstream language and requires some learning, but it has always been the primary language for Cocoa development. Java is more mainstream, but it uses a Java bridge to talk to the Cocoa framework—and this process can really slow down your program. According to Aaron Hillegass, author of *Cocoa Programming for Mac OS X:*

> The Java bridge is a wondrous piece of software. Enabling developers to write Cocoa apps in Java was a hard problem, and the programmers who created the bridge are among the best in the world.…
>
> That said, I would never advise a client to write a Cocoa application in Java. Cocoa is a very elegant framework when used with Objective-C. Everything becomes buggy, slower, larger, and less documented when you write it in Java.[1]

For fairness, make sure you read his explanations and supporting examples to get the whole picture.

---

[1] http://www.bignerdranch.com/Resources/Java.html.

5.3 Cocoa software infrastructure

Cocoa is a collection of object-oriented libraries, also called *frameworks*, which enable developers to construct GUI-based applications for Mac OS X. You interact with the Cocoa frameworks in either Java or Objective-C. In addition to the frameworks, Cocoa contains a runtime system that runs Cocoa applications. One way to think of Cocoa is as a collection of software integrated circuits (software ICs) that you have access to and can use to construct aspects of your program. Cocoa is not a new technology designed exclusively for Mac OS X; it's been around since the days of the NeXT computer.

Cocoa is divided into two subsystems: the *Foundation* and the *Application Kit*. The Foundation classes implement non-GUI classes that act as utility classes for an application. The Application Kit contains classes that provide developers with the basic GUI functionality required by most applications. Let's take a high-level look at the components of the Foundation, and examine a few examples of how to use the classes.

5.3.1 Foundation

Foundation is composed of several kinds of classes including value objects, strings, collections, operating system services, file system, interprocess communication, threading, scripting, and distributed objects. See http://developer.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/ObjC_classic/IntroFoundation.html for a discussion of the Foundation class hierarchy.

Let's look at a few examples of how to use these classes.

NSString

Most object-oriented languages supply a class for storing and manipulating strings. String classes let you store an arbitrary length string without worrying about storage details. In addition, the string class implements methods that cover the space of possible string manipulation operations, such as getting the length of a string, concatenating two strings, and equality operations. C++ provides these facilities through the `string` class, and Java uses the `java.lang.String` class. Cocoa provides string operations with the `NSString` and `NSMutableString` classes.

The `NSString` class operates on immutable strings, or strings that once created, cannot be changed. `NSMutableString` handles mutable strings, which are strings that can and often change after creation. In both cases, the string classes store strings as Unicode characters.

The simplest way to declare a string is as follows:

```
NSString *firstStr = @"My first string!";
```

Here, the @ character indicates that the string is a string object constant.

NSString supports the formatting of strings in a way similar to C's sprintf function. For example, the following C code

```
char aStr[256];
int x = 10;
sprintf(aStr, "This is a formatted string with an int %d.\n", x);
```

is written as follows in Objective-C:

```
NSString *aStr;
aStr = [NSString stringWithFormat:
       @" This is a formatted string with an int %d.", x];
```

In both cases, the call sets the string aStr to the new formatted string.

Objective-C builds on C, so you may come across cases where you need to convert between a C string and an NSString. The following example shows how to perform this conversion:

```
char cStr[256] = "A String";
// Convert a C string to an NSString.
NSString *nsStr = [NSString stringWithCString: cStr];
// Convert the NSString back to a C string.
char *p = [nsStr cString];
```

The NSString class has a static method called stringWithCString that takes an array of characters as a single parameter. Additionally, NSString has an instance method, cString, which returns a C string in the default C string encoding. Objective-C automatically handles deallocating the allocated memory for the string when it is destroyed. In section 5.3.3, you learn more about Objective-C's deallocation mechanism. NSString also supports the usual assortment of operations such as string comparison, string length, and string access functions.

The writeToFile method writes the contents of the receiver (NSString object) to the file specified by the path string. The parameter flag is set to either YES or NO. If YES, it writes the string to a temporary file and, if successful, writes the temporary file to the file specified by path. If NO, it writes the string directly to the file named by the path parameter. The writeToURL method performs a similar operation, but writes to a URL:

```
(BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag
(BOOL)writeToURL:(NSURL *)anURL atomically:(BOOL)atomically

NSString *buf = @"Bound for a file";
[buf writeToFile: @"/Users/omalley" atomically: YES];
```

To read a file into an NSString object, you use the stringWithContentsOfFile method:

```
NSString *fileBuf;
NSString *fileName = @"/home/omalley/data.txt";

fileBuf = [NSString stringWithContentsOfFile: fileName];
if (string == nil) {
  // Handle error...
}
```

As you can see, the `NSString` class implements fundamental operations for manipulating strings in Objective-C. See the `NSString` documentation for more information about these and other operations (http://developer.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/ObjC_classic/Classes/NSString.html#//apple_ref/occ/instm/NSString/cString).

NSDictionary

Collection classes are fundamental in most languages, class libraries, and application frameworks. Collections, sometimes called *containers*, provide developers with a set of classes for storing and accessing application-defined data. For example, C++ supports collections through the *Standard Template Library* (STL) (actually, STL provides more support, including iterators and algorithms). No matter the implementation, the principles are the same: to provide a set of classes that enables the efficient storage and access of application data.

Suppose you are writing an auction program for an online e-commence site. Auctions typically work as follows: read and store a sequence of bids from users and periodically generate price quotes and clears using a matching algorithm, based on the auction rules. At the heart of this sequence is the matching algorithm, which defines what bids generate trades. The rest is mainly software infrastructure and involves storing and accessing bids. For example, there are many implementations for storing a sequence of bids, including placing them in a list, vector, or a hash table. Lists are easy to implement, but they are accessed in linear time `O(n)`. A better choice is a hash table, which exhibits a constant time complexity `O(1)`. If you assume that each bid is associated with a user ID, that user IDs are unique, and that auctions can have only one active bid, then hashes are a good design choice.

ASYMPTOTIC COMPLEXITY One of the most common benchmarks for evaluating an algorithm's performance is the *asymptotic complexity measure*. Asymptotic complexity measures how an algorithm performs in relation to the size of its input. This measure lets you express how a given algorithm will perform independent of platform issues such as compilers or machine architectures.

This measure is expressed using *O-notation*, which is commonly called *big-O* notation. The *O* is read as *order*, and the notation reads as follows: it will take on the order of *K* steps to perform the algorithm. For example, it takes on the order of `O(n)`, steps to perform a sequential search of an array, or `O(log n)` to perform a binary search.

The follow table shows common measures from best to worst performance:

O-notation	Name
`O(1)`	Constant
`O(log n)`	Logarithmic
`O(n)`	Linear
`O(n log n)`	No name (usually called (n log n))
$O(n^2)$	Quadratic
$O(n^k)$	Polynomial
$O(2^n)$	Exponential

In C++, a possible implementation follows in listing 5.1.

Listing 5.1 Storing bid information in a hash table

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

map<int, string, less<int> > activeBids;

void
AddBid(int ownerID, char * const bidStr) {
  if (bidStr == NULL)
    return;
  string s = bidStr;
  activeBids [ownerID] = s;
}

int
main (int argc, const char * argv[]) {
  map<int, string> bids;

  AddBid(1, "1,1,12.99");
  AddBid(3, "3,1,42");
  AddBid(7, "4,1,3.14");
```

```
  // Print the result.
  for(map<int, string, less<int> >::iterator
      it = activeBids.begin ();
      it != activeBids.end(); it++) {
    cout << "Key: " << (*it).first
         << " for value: " << (*it).second << endl;
  }
  return 0;
}
```

In this example, bid information is stored in a hash table, whose key is the user ID. The Cocoa Foundation collections give you this functionality though the `NSDictionary` and `NSMutableDictionary` classes. Listing 5.2 shows a possible bid storage and access implementation using the `NSMutableDictionary` class.

Listing 5.2 Using NSMutableDictionary as a container for bids

```
// auction.h
#import <Foundation/Foundation.h>

@interface auction : NSObject {
  NSMutableDictionary *activeBids;
}

-(BOOL) addBid:(NSString *) ownerID: (NSString *)bidStr;
-(void)print;
@end

#import "auction.h"
// auction.m
@implementation auction

- (id) init {
  self = [super init];
  activeBids = [[NSMutableDictionary alloc] init];

  return self;
}

- (void) dealloc {
  [activeBids release];
  [super dealloc];
}

-(BOOL) addBid:(NSString *) ownerID: (NSString *)bidStr {
  if ( (ownerID == nil) || (bidStr == nil) )
    return NO;

  [activeBids setObject: bidStr
                forKey:  ownerID];
```

```
  return YES;
}


-(void)print {
  NSArray *keys;
  id key, value;
  int i;

  // NSLog prints debug statements.
  keys = [activeBids allKeys];
  NSLog(@"Size: %d", [keys count]);
  for (i=0; i<[keys count]; i++) {
    key = [keys objectAtIndex: i];
    value = [activeBids objectForKey: key];
    NSLog(@"Key: %@ for value: %@", key, value);
  }
}

@end

#import <Cocoa/Cocoa.h>
#import "auction.h"

int
main(int argc, const char *argv[]) {
  // NSAutoreleasePool implements Foundation's autorelease mechanism
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  auction *anAuction = [[auction alloc] init];

  [anAuction addBid:@"1":@"1,1,12.99"];
  [anAuction addBid:@"3":@"3,1,42"];
  [anAuction addBid:@"4":@"4,1,3.14"];

  [anAuction print];

  [anAuction release];
  [pool release];
  return 0;

}
```

By using the Foundation container classes, you avoid implementing your own collection classes for storing and accessing the bids (which is time consuming and error prone).

5.3.2 Application Kit

Cocoa's Foundation classes provide your application with fundamental support for common internal program operations such as storing and accessing data structures. These operations are basic to an application's functionality and are

used but not directly seen by users. The Cocoa Application Kit supports the visible aspects of your application. The Application Kit consists of a set of classes that provide application developers with infrastructure for developing the user interface of an application, including windows, menus, controls, buttons, and text fields.

The Application Kit includes more than 100 classes, but as the Apple documentation points out, you can access the classes at different levels of complexity:

- At its highest level, you interact with the Application Kit through Interface Builder. In this mode, you use Interface Builder to draw your user interface, and you write handler routines in Project Builder that respond to user actions.
- You can interact with the Application Kit and classes on a more detailed level by dealing with the framework more directly and writing code to handle more advanced user interactions with your application's user interface. For example, you might write code to handle copying and pasting text or drawing lines or shapes in an application window.
- The lowest level of interaction involves deriving new classes based on parent Application Kit classes. At this level, you are specializing classes to add functionality to existing framework classes.

See http://developer.apple.com/techpubs/macosx/Cocoa/Reference/ApplicationKit/ObjC_classic/IntroAppKit.html for a discussion of the Application Kit class hierarchy.

5.3.3 Memory management

Much of programming and program design requires managing complexity. As a program grows, as you add more features, or as you implement more advanced algorithms, the complexity of the program often grows, requiring you to manage increasing numbers of details. For any program, one of the most important details is memory management. Depending on your development language, memory management can be straightforward or quite detailed, requiring you to account for every allocated memory block. Improper tracking of allocated memory can result in poor program performance, incorrect behavior, and program crashes.

To address these issues, many programming languages include built-in support to help programmers efficiently manage memory. There are three main categories of language-based memory management support:

- Offloading management issues to the programmer, as demonstrated by the C language

- Providing language-level support functions and infrastructure that track memory and help the programmer supervise memory allocation and deallocation (Objective-C)
- Handling a program's memory management through a runtime garbage collector, thereby freeing the programmer from worrying about memory management details (LISP, Java)

Let's look at these schemes, focusing on Objective-C's memory management and what you need to know to use it effectively.

C

Memory management in C reflects its language design: provide the programmer with an expressive and powerful language that can be applied to general application development as well as system programming. It should put the power in the hands of programmers and stay out of their way as much as possible. As the saying goes, it gives you all the power to shoot yourself in the foot (as opposed to C++, which will let you take off the entire leg). This design translates into a general-purpose memory allocation package that provides basic memory-management functions; after that, you are on your own.

The C language implements memory operations in the `malloc/free` family of memory allocation/deallocation calls. Over the years, hosts of C-based support tools have evolved to aid programmers in checking and debugging memory allocation in their program. They include static code checkers like `lint`, the popular `Pslint`, custom memory allocation packages like `debugmalloc`, and runtime analysis tools such as Purify that detect a variety of memory errors at runtime.

The advantage of C memory management (or lack thereof) is performance. The disadvantage is that the programmer must be extremely careful in handling memory management, which is notoriously error prone.

Garbage collection

At the other end of the spectrum are languages that support runtime garbage collection. The design and implementation of garbage collection systems is a big topic; in short, a garbage collection system (garbage collector) is responsible for tracking memory allocations throughout a program's lifetime and reclaiming memory when it is no longer needed. The programmer is not responsible for keeping track of memory as the program runs or making sure it is deallocated.

Once again, this method has advantages and disadvantages. The programmer does not need to track and manage memory, and is assured that the program will not malfunction because of memory-related problems (well, almost assured). The

disadvantage is performance: garbage collectors eat CPU cycles and will place a drain on your program's performance. Design decisions, like life, are a tradeoff.

Reference counting in Objective-C

Somewhere between these systems lies Objective-C's—and, by extension, Cocoa's—method of memory management. The basis of memory management under Cocoa is a technique called *reference counting*. Reference counting is conceptually quite simple: it works by tracking the number of references that exist to an object. Each time there is a new reference, the reference count is incremented by one; when a reference is removed, the count is decremented by one. When the count reaches zero, the object is no longer in use, and the runtime system can safely free the memory.

Cocoa accomplishes reference counting through a few memory-related calls, which are described in table 5.2.

Table 5.2 Cocoa handles basic memory management through a technique called reference counting.

Method name	Description
alloc	Allocates memory and returns an instance of the allocated object. Sets the reference count for the object to one.
release	Decrements the receiver's reference count by one. When its reference count becomes zero, it sends the receiver a dealloc message. The dealloc message frees memory held by the receiver.
autorelease	Adds the receiver object to the current autorelease pool. At some later stage, the pool operations decrement the receiver's reference count by one.
retain	Increments the receiver's reference count by one.
copy	Copies the object and set its reference count to one.

Now that you understand the basics, let's look at some examples of how reference counting works using the Cocoa memory methods. We'll begin with a simple memory allocation and deallocation example. In the following listing, the code allocates and releases a string. Between calls, it prints the string and its reference count, which in this case equals one:

```
-(void)simpleAllocDealloc {
 NSString *s = [[NSString alloc] initWithString:@"test string"];
 NSLog(@"string object '%@' has reference count of %d.\n",
    s, [s retainCount]);
 [s release];
}
```

This example demonstrates how to perform a basic memory allocation operation using the `alloc` and `release` methods. The `alloc` method creates a new block of memory for the object, sets the receiver's reference count to one, and returns the memory to the caller. The `release` method decrements the receiver's reference count by one and sends the receiver a `dealloc` message if the count equals zero.

The next listing illustrates how to return an allocated object to a caller. The `NSString` object is valid only within the scope of its declaration—in this case, only within the function. However, suppose you need to implement a method that returns an object the method allocated within its scope:

```
-(NSString *)returnAllocString {
  NSString *s = [[NSString alloc] init];
  s = @"test string";
  return s;
}
```

You need a mechanism that retains the object until the caller is done with it, but that guarantees the object is deleted. Remember, each time you allocate memory, you must include a corresponding instruction that deallocates the memory.

To address this issue, you use an *autorelease pool*. At the beginning of the application's event loop, the runtime system has an autorelease pool. As the event loop runs, it calls application code. When an application is done with a block of memory, it sends an `autorelease` message, which adds the object to the autorelease pool. At the end of the current iteration of the event loop, the application object sends a `release` message to the autorelease pool, causing the autorelease pool to send a `release` message to each object in the pool. If the reference count of any object in the pool is zero, the object is deallocated. This cycle continues until the application terminates. This mechanism implements the semantics of the `autorelease` method, which says to delete the object you send the message to at some later time (*later* being at the end of the event loop).

Here's a function that uses the `autorelease` method and autorelease pool to ensure that memory is properly deallocated in the caller code:

```
// Foo.h
#import <Foundation/Foundation.h>

@interface Foo : NSObject {
}

-(NSString *)returnAllocString;

@end

// Foo.m
#import "Foo.h"
```

```
@implementation Foo

-(NSString *)returnAllocString
{
  NSString *s = [[NSString alloc] initWithString:@"test string"];
  return [s autorelease];
}
@end

// main.m
#import <Cocoa/Cocoa.h>
#import "Foo.h"

int
main(int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  Foo *foo;
  NSString *s;

  foo = [[Foo alloc] init];
  s = [foo returnAllocString];

  [foo release];
  [pool release];
  return 0;
}
```

By using the autorelease pool through the `autorelease` method, you make sure the caller gets the memory object it expects and that the memory is marked for proper deallocation. Remember, the `NSApplication` class sets up and initializes a Cocoa application autorelease pool. If you are writing an Objective-C program that does not use Cocoa (Foundation Tool; you choose this from the New Project list after selecting File→New Project), make sure you choose Foundation Tool when you create the new project.

In addition to using the Application Kit's built-in autorelease pool (set up and initialized in `NSApplication`) or adding it to the main function of non-Application Kit programs (Foundation Tool), you can create local copies of an autorelease pool that operates on a per-method or scope basis. The advantages of a local autorelease pool are performance and memory size; the main application pool does not grow too large and therefore does not take excessively long to deallocate pooled objects.

Objective-C's reference counting gives you better control over handling memory allocation within your program. This scheme is not perfect, but by understanding the basic design of reference counting and its use, you can exert greater control over memory management in your program and produce more manageable and verifiable code.

5.3.4 Design patterns

Objective-C and Cocoa help programmers write better software by embodying many common and useful design patterns. Design patterns are proven methods for building better and more reliable programs, and are important to understand when developing Cocoa programs.

Over the years, software engineers have developed many commercial software systems—some very good, others not so good. The good ones succeed for many reasons, including experienced management teams, reasonable schedules, stable software development practices, and solid software designs based on proven models. Conversely, unsuccessful projects exhibit the inverse of many of these traits. Many would argue that when you develop new systems, you should base them on the best designs of successful projects. Moreover, if the system is designed right, you can reuse parts of the system in future systems, thereby decreasing overall development time and risk for future projects.

When building new systems, there's a real difference between theory and practice. For example, how many times have you developed a system that iteratively collects data, transforms it in some way, and displays the transformed data, possibly in various formats? What if each time you developed a program that required this functionality, your design was different? This process would result in constantly reinventing the wheel and writing code that was not reusable in other projects, and would be a real waste of your time and your company's money.

To address these issues, *design patterns* decouple the design from the domain and let you use a set of the most useful, general, and applicable designs as a basis for new software:

> Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together. The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is

> on creating a culture to document and support sound engineering architecture and design.[2]

The wins you get by understanding and applying design patterns in your daily work include simplicity of design, software based on solid and proven designs, and a good step toward maximizing reusability.

As much of the Apple documentation on Cocoa points out, design patterns play an important role in the design of the Cocoa frameworks, as well as Cocoa programs. In Cocoa, four primary design patterns emerge: Model-View-Controller (MVC), Target/Action, Delegation, and Chain of Responsibility. Let's look briefly at these patterns and see how they apply to Cocoa.

Model-View-Controller (MVC) pattern

The MVC pattern can be traced back to the days of designing interfaces in Smalltalk. Strictly speaking, the MVC pattern comprises three groups of classes, sometimes called the *MVC triad:*

- The *model* holds data describing the state of the application. It responds to requests to update its state and returns its data to clients. The model is directed by the controller and sends update messages to the view in response to state changes.
- The *view* is responsible for displaying the data contained in the model. An application can have more than one view, providing the user with different views of the model. Each view is controlled either by a single master controller or possibly by different controllers; it receives update messages from the model to update its display when state in the model changes.
- The *controller* acts a mediator between the model and view, and routes application requests from either a user or device to the view and data.

Figure 5.1 shows an example of the MVC pattern.

Overall, the MVC pattern describes a generic, reoccurring design that you can apply to many programs. It makes a clear separation between program components and their responsibilities and enables you to reuse components of the program (specifically, the model and view) in other programs with little or no code modification. The controller is typically specific to an application.

Under Cocoa, the MVC pattern is a useful way to structure an application. Cocoa defines a number of view objects that programs reuse to display application

[2] http://hillside.net/patterns.

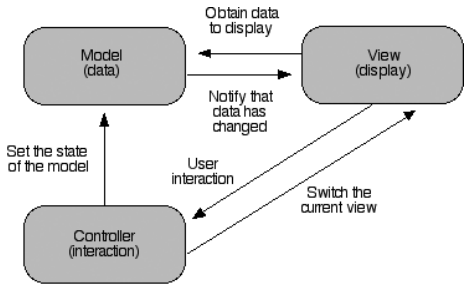

Figure 5.1
The MVC pattern describes a design that can be applied to many problems. It enables a clear separation between program components, emphasizes reuse of the model and view components, and lets you easily add many views to a program.

data and post messages when a view is changed. As recommended in O'Reilly's *Learning Cocoa* book, new Cocoa core features like the document architecture, undo support, and scripting are simpler to use if your programs design follows the MVC pattern.[3]

Target/Action (command) pattern

User interfaces provide a simple, intuitive way for users to interact with a program. Application frameworks encapsulate much of the infrastructure for handling user events and provide developers with primitives for building user interfaces. On top of these frameworks, you implement specific code that responds to user actions. The Target/Action pattern, called the Command pattern by Eric Gamma, defines a general method for a framework to make requests for services implemented in an application, where the framework has no knowledge of the application objects.[4]

Cocoa implements this pattern in its handling of framework controls and actions. For example, imagine a user of your application clicks on an interface control—say, a button. This click event generates a message, which the framework sends to your application. Application-level code implements how the application responds to this action. The sending of the action message is in the framework; the response is in the application. The framework does not know how to respond to the message; it only knows that the event occurred. The Target/Action pattern defines a general way an object can send messages to an undetermined object.

In Cocoa, this pattern is set up in Interface Builder and implemented by handling code in Project Builder. When you create a new control, class, and instance in Interface Builder and Control-drag from the control to the instance, you are defining a target action.

---

[3] Apple Computer Inc., *Learning Cocoa*, ed. Troy Mott (Sebastopol, CA: O'Reilly, 2001).

[4] Erich Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series (Reading, MA: Addison-Wesley, 1995).

Delegation pattern

Delegation in object-based systems is a powerful way to handle the problem of extending an object's functionality without inheritance (see section 5.2.1 if you need to refresh your knowledge of OO terminology).

Delegation lets you achieve some specialization through code reuse. As Gamma et al point out, inheritance enables a *being* relationship between a parent and child class, whereas delegation is a *have* relationship: one class would have or contain another class (this is sometimes referred to as an *is a* or *has a* relationship).

You implement delegation by having the main class keep a pointer to the delegation class instance, which it uses to access methods in the delegation class. Instead of inheriting operations from a parent, the class passes requests to its delegate through its pointer. Other design patterns use delegation, including the State, Strategy, and Visitor patterns (see Gamma for a description of how delegation is used in these patterns).

Cocoa implements delegation using Objective-C's delegation services. For example, to implement it in Interface Builder, follow these steps:

1. Create two new classes: one for the delegate class (`MyDelegate`) and one for the class that holds the delegate pointer (`MyHolder`).
2. Add an outlet (data member) to the `MyHolder` class for the delegate class pointer and create the `MyHolder` class files and instance.
3. Open the Instances pane and Control-drag *from* `MyHolder` *to* `MyDelegate`, select the outlet that holds the pointer, and click the Connect button.
4. Create the files and instance for `MyDelegate`. You will need to implement the delegation code in the delegate class from within Project Builder.

Chain of Responsibility pattern

Object-based systems define properties and behavior in classes, which are instantiated into objects at runtime. As a program runs, its objects interact by sending messages to one another that request services or perform a particular action. In some cases, the sender explicitly knows what object should handle its request. For example, the following code sends a display message to the receiver object:

```
[myPictView display];
```

In this case, the caller knows the receiver object. However, what if the caller does not know who should handle the message, but would rather send the message to a set of objects and let them decide who should handle the request?

Let's relate this question to a real-world example. Imagine you head a development group consisting of three teams. The first team is the least experienced and handles basic coding issues. The next team handles issues that are more advanced, and the third team is responsible for designing and implementing advanced features. You need a new feature added to the program, so you send a message to the first team. They discover that the addition requires more experience than they possess and forward the message to team two, who determine that it will require some design changes as well as more advanced coding experience. They forward the request to the third team, who implement the feature. In this case, you sent the original message with the understanding that the team best able to implement the feature should do so; you really do not care what group performs the implementation. Effectively, the teams form a *chain of responsibility*, where each is responsible for either handling requests they are suited for or forwarding the request along the chain.

This example demonstrates the basic principles of the Chain of Responsibility design pattern, which breaks the link between the sender of a message and the specific object that will handle the request. It replaces this link with a more general semantic that says the message should be handled by the most capable object in the chain.

This pattern is used in the Cocoa frameworks in its handling and routing of messages to windows and views. For example, if you click on an active object (say, a button) in a view, that view becomes the first responder. If that view contains a method to handles the event, it handles the request. Otherwise, it passes the event to the next object in the chain. The next object either handles the request or forwards it up the chain. This process continues until one of the objects handles the message or the message falls off the end of the chain and is not handled.

5.3.5 Cocoa event handling

From a user point of view, a Cocoa application looks pretty much like any other Mac OS X program. It contains menus, windows, and dialog boxes that work together to respond to user commands that perform some task. Users do not see that the visible components of a program are the facade over a much more complex interaction of Cocoa interface objects (Foundation, Application Kit, and user classes); Quartz, which handles drawing windows and graphics; and the underlying operating system.

An event-driven system works as follows (I will exclude events generated by system tasks like timers and concentrate only on user-generated events):

1. A user interacts in some way with an application: pressing a key to enter text, clicking the mouse on a menu item, or perhaps clicking a button control.
2. The system reads the event and sends it to the application that should service the event, where it is placed on the application's event queue.
3. At the appropriate time, the application's event loop pulls the top event off the queue and routes it to the proper handling routine within the application.

In Cocoa, the event loop is called an *event cycle*. Each event on the event queue is stored in the `NSEvent` object, which encapsulates information that describes an event. For example, an event holds general information such as the type of event and the time the event was generated. For keyboard events, extra information is stored, including the character the user pressed and its key code. For mouse events, the event holds the location of the mouse at the time of the event and the identity of the mouse button that generated the event. See the class documentation for more information (http://developer.apple.com/techpubs/macosx/Cocoa/Reference/ApplicationKit/ObjC_classic/Classes/NSEvent.html); but, in general, anything you need to know about an event is stored in this object.

My description of the Cocoa event cycle is quite general and does a fair amount of hand waving. Usually, Cocoa applications accomplish event handling through the Chain of Responsibility pattern (see the preceding section). This pattern generalizes message handling by breaking the link between a message sender and receiver. Rather than sending a message to a specific object, an object sends it to a set of objects (in Cocoa, called the *responder chain*) and assumes the object that should handle the request will either do so or pass the request up the responsibility chain. This process is quite powerful and has many advantages over one-to-one message passing.

Let's look at the interaction of event messages and the responder chain. Conceptually, chain of responsibility, or the responder chain, works as follows:

1. The `NSApplication` object pops the next event from its event queue and sends the message, using its `sendEvent` method, to the window associated with the event.
2. The window object (`NSWindow`) sends the event to the first responder, typically the currently selected view (`NSView`).
3. If this object can handle the request, it does; otherwise, it passes the event to its next responder, and so on up to the `NSWindow` object.

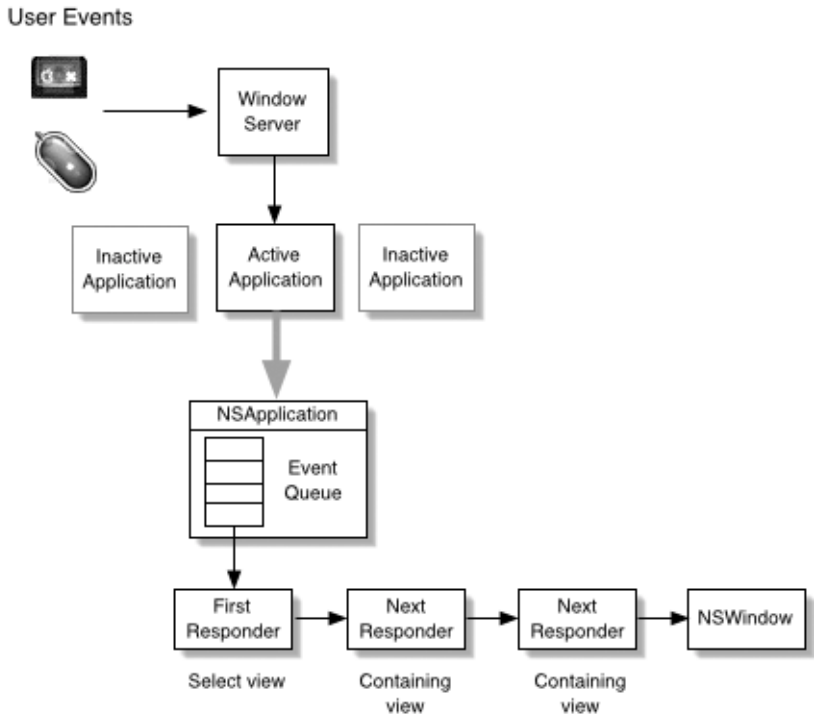

Figure 5.2 User events are forwarded by the window server to the active application, where they are placed in the application event queue and sent to the window's responder chain.

4 If no object can handle the event, the last responder object calls its `noResponderFor` method to handle the condition. A view's next responder is its `superview`. Figure 5.2 demonstrates the Cocoa event cycle.

To summarize, events go from a view's first responder to the next responder (its `superview`) and so on up to the window. If any responder object handles the event, control returns to the `NSApplication` object.

This discussion has given you a flavor of how Cocoa handles events. However, Cocoa's use of responder chains is far more complex than I've described here and includes such elements as action events and key windows. Much of this detail and interaction will become clearer as you write more Cocoa application and dig into the details of how your application interacts with these elements. For more information about these topics, see Apple's documentation on event handling and the responder chain (http://developer.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/AppEventHandling/AppEventHandling.html).

Event tracing

Before concluding this discussion, let's look at how to enable event tracing in your Cocoa programs. During development, it is sometimes useful to see the stream of events that come from the window server to your application and are popped from the event queue. Cocoa provides this facility by setting the `NSTraceEvents` flag. There are a few ways to use this feature. One is to open a shell, change to the directory that holds your Cocoa program, and run the program as follows:

```
% ./NSTraceEventsCocoaExample -NSTraceEvents YES
2002-03-30 09:25:00.869 NSTraceEventsCocoaExample[16706]
timeout = 63074799299.132416 seconds, mask = ffffffff,
dequeue = 1, mode = kCFRunLoopDefaultMode
2002-03-30 09:25:00.894 NSTraceEventsCocoaExample[16706]
got apple event of class 61657674, ID 6f617070
2002-03-30 09:25:00.900 NSTraceEventsCocoaExample[16706]
still in loop, timeout = 63074799299.100403 seconds
2002-03-30 09:25:00.902 NSTraceEventsCocoaExample[16706]
timeout = 63074799299.100403 seconds, mask = ffffffff,
dequeue = 1, mode = kCFRunLoopDefaultMode
2002-03-30 09:25:03.169 NSTraceEventsCocoaExample[16706]
Received event: Kitdefined at: 0.0,0.0 time: 171471 flags:
0 win: 35733 ctxt: 0 subtype: 9, data: 1e50,0
2002-03-30 09:25:03.173 NSTraceEventsCocoaExample[16706]
Received event: LMouseDown at: 515.0,156.0 time: 171471
flags: 0 win: 35733 ctxt: 127d7 data: -25994,1
2002-03-30 09:25:03.175 NSTraceEventsCocoaExample[16706]
In Application: NSEvent: type=LMouseDown loc=(328,260)
time=736464.6 flags=0 win=0 winNum=35733 ctxt=0x127d7
evNum=-25994 click=1 buttonNumber=0 pressure=1
```

Remember, Cocoa applications are stored in a bundle, so you will need to change to the appropriate directory that holds the executable program—usually something like ~/[project-directory]/build/[program-name].app/Contents/MacOS.

Another technique is to set `NSTraceEvents` within Project Builder so tracing information is displayed in the Run pane. To accomplish this, open your project in Project Builder, select the Executables tab, and select the program from the list. Next, click on the plus icon under the Arguments category and add the launch argument shown in figure 5.3.

Next time you run the program, Project Builder will write event-tracing data to the Run pane.

5.4 Other Cocoa development languages

Apple officially supports two programming languages for developing Cocoa applications: Objective-C and Java. However, designers of several projects are

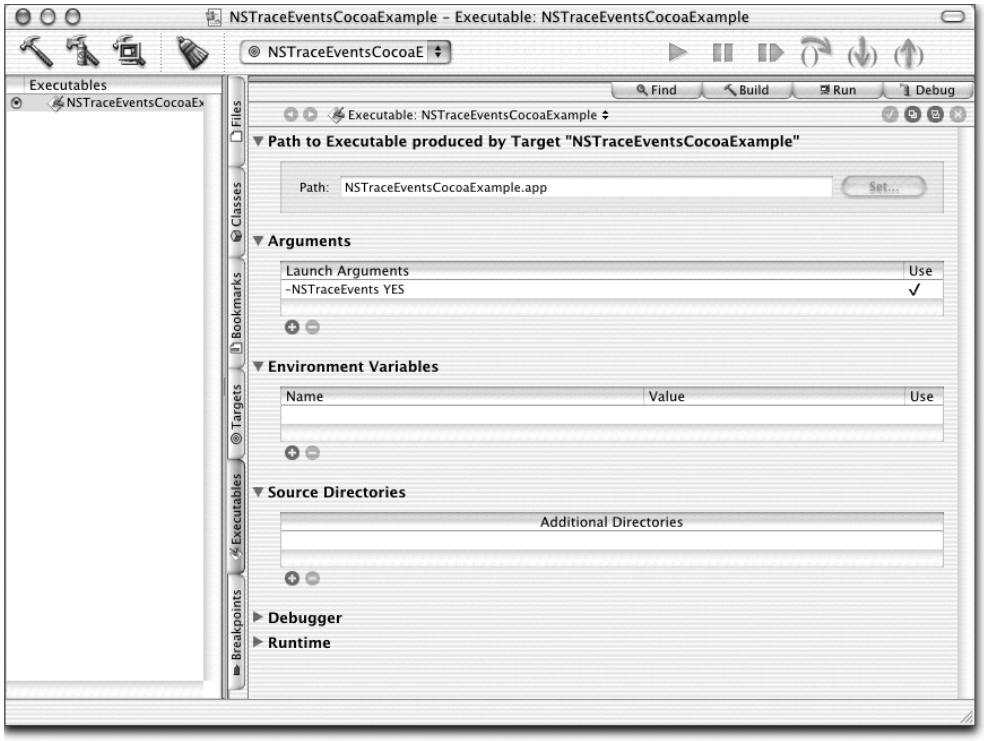

Figure 5.3 Setting the `NSTraceEvents` to YES in Project Builder enables the display of event messages in the Run pane.

working to bring other languages to the table so developers can get the advantages of the Cocoa frameworks in the language of their choice. Be forewarned: most of these projects are early in the development process and do not support the full feature set you get under Objective-C and Java.

5.4.1 C++

C++ is not currently supported for developing Cocoa programs. However, mixing C++ code with Objective-C is legal, and you can do so under Project Builder and its supporting compilers. See the Big Nerd Ranch site for an example of mixing C++ and Objective-C (http://www.bignerdranch.com/Resources/Examples.html).

5.4.2 Perl

As most UNIX developers already know, Perl is a great language for processing text files, writing to CGIs, or developing network programs. Having a Perl bridge to

Cocoa would be very useful for putting interfaces on Perl scripts, and would give you a quick prototyping tool.

Jaguar contains a Perl module called PerlObjCBridge, which bridges Perl and Objective-C runtimes. This addition is exiting for Perl developers who wish to use Perl and Cocoa. Unfortunately, this version of PerlObjCBridge does not support writing GUI Cocoa applications in Perl, but the CamelBones project does; it even provides a Perl Project Builder template for developing Perl-based Cocoa programs (http://www.sourceforge.net/projects/camelbones). Chapter 8 discusses both PerlObjCBridge and CamelBones.

5.4.3 Ruby

Ruby, created by Yukihiro Matsumoto, is a relatively new, freely available object-oriented scripting language. Ruby supports text-processing functionality similar to that provided by Perl, and also supports network programming. For more information about Ruby, see http://www.ruby-lang.org/en.

RubyCocoa is a combination Mac OS X framework and Ruby library; its project goal is to let programmers use Cocoa objects through Ruby scripts. For more information about the RubyCocoa project, see http://www.imasy.or.jp/~hisa/mac/rubycocoa.

5.5 Summary

This chapter has introduced you to Objective-C and Cocoa. I've presented the basics of the Objective-C language, shown you how to read Objective-C code, and covered its memory management scheme. You've also learned the fundamentals of Cocoa, Apple's object-oriented framework for developing Mac OS X applications. Cocoa is composed of the Foundation and Application Kit frameworks, each of which provides developers with a different software infrastructure for developing Mac OS X programs.

The Cocoa frameworks, as well as programs written using Cocoa, use many design patterns that let developers use proven design techniques in their Cocoa applications. In addition, you have seen how events are handled in Cocoa programs. Finally, I discussed how languages in addition to Java and Objective-C are in the works for developing Cocoa programs.

In chapter 6, you'll put this knowledge to work as you build your first real Cocoa program in Objective-C.

6 Cocoa programming

- Designing a Cocoa program
- Building a program's GUI Interface Builder
- Creating classes, class instances, and actions
- Writing code in Project Builder
- Running UNIX command-line tools as subtasks

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

—Rich Cook

In Chapter 5, I discussed Cocoa, Apple's object-oriented framework for developing Mac OS X applications in Objective-C and Java. You learned about Cocoa, its software infrastructure, and the main concepts you need to know before writing code for Cocoa. This chapter takes you through the steps of developing a fully functioning Cocoa application. The program is a GUI front end for `wget`, the GNU command-line network utility that retrieves files and directories from the Web over Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP). After reading this chapter, you will be well on your way to using the Cocoa framework to write your own applications.

6.1 Introduction

Centralizing an application's features to a single program running in one address space is a common and straightforward design, but it can be somewhat limited. For example, imagine you are writing an auction server that operates in a market environment that supports a number of different auctions. Depending on the auction (CDA, Vickrey, and so on), a specific set of operations needs to be performed—some specific to the type of auction and some general to all auctions. Bid processing, clearing, and information revelation (quotes) are part of the auction logic. Getting bids from clients, queuing the bids, and passing them to the auction are general operations, common to all auctions.

Think of eBay (http://www.ebay.com). Users submit bids to an auction through a web browser, and the bids are stored on the eBay site. An eBay auction processes its bids and posts intermediate and final results to its web site, which users view through their web browser. Getting and storing bids from users is completely independent from the auction logic.

You could design your auction server by abstracting all the common infrastructure to a common location—say, a class, which is used by all auctions. In this design, operations like reading bids from multiple clients, queuing bids, and passing bids to the auction logic are packaged in one or more classes, which are used by all auctions. Another implementation, which is more distributed and language independent, runs the common infrastructure code in a separate process; auctions interact with it through an interprocess communication (IPC) mechanism such as

TCP sockets. In either case, separating the common from the specific is good design and enables developers to build systems more quickly and application safely.

You can scale this example down to applications that run on your desktop machine. In this case, being able to decouple a program interface from its working code has some useful properties. You can write your main program logic in a command-line program (or perhaps use an existing UNIX command-line program) and write a GUI that enables access to the program features. Using this approach, you can also target your program to different types of users. UNIX users who prefer the command-line environment will run the command-line version of the program. Those who prefer GUI-based interfaces can operate the program through its GUI. This powerful technique is becoming popular among many Mac OS X developers.

6.2 The CocoaWGet example program

The example program you'll build in this chapter is called CocoaWGet. It is a Cocoa front-end for `wget`. If you are already familiar with `wget`, you know what a valuable and powerful program it is. If you have never used it before, you are in for a treat. With `wget`, you can download the contents of a web site (including graphic and sound files) to your local machine with a few simple commands.

For example, imagine you need to mirror a remote web site on your local machine, or perhaps you come across a site that contains a collection of images or sound files you want. With `wget`, grabbing these files is painless. I typically use `wget` to download papers and articles from sites for offline reading or archiving.

The following are some typical `wget` commands for performing retrieval operations:

```
# Gets index.html from the URL
wget http://www.site-o-interest.org
# Gets the specified file from site over ftp
wget ftp://ftp.site-o-interest.org/file-to-get
# Recursively get files from the site, saving to /tmp
wget –r –P/tmp http://www.site-o-interest.org
# Get files from the sites listed in url-file
wget -i url-file
```

As you can see from these examples, using `wget` from the command line is straightforward. However, `wget` has roughly 70 options. If you're like me, keeping track of the basic commands is easy, but I have to look up the subtle ones each time. Adding a Cocoa interface to `wget` makes the program options easy to locate, and also makes the program accessible to users who are not comfortable with the UNIX command line. In addition, it is a nice example of how you can leverage the power of existing programs to create new programs—something UNIX people do all the time.

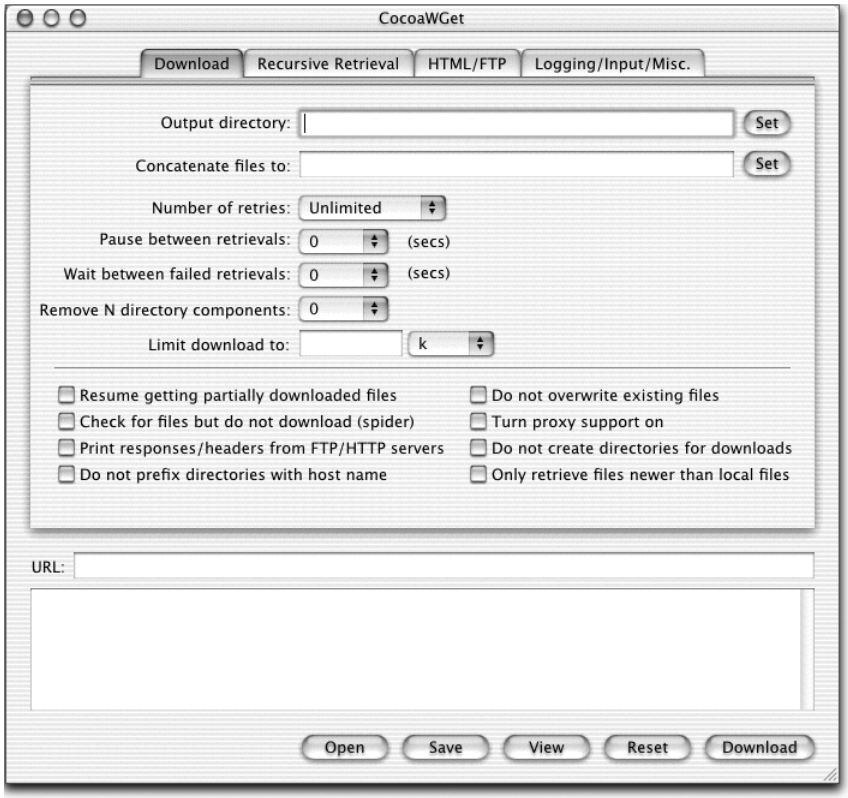

Figure 6.1 CocoaWGet provides a tabbed control interface for accessing `wget` options.

Figure 6.1 shows the Cocoa-based GUI for CocoaWGet. The program gives you complete access to the `wget` command-line options through a series of tab controls. Once you select a set of `wget` options, you click the Get button, causing the program to collect the selected options into a `wget` command line and run the `wget` program. Clicking Reset initializes the interface to its default values. The View button displays the current command line based on the state of the interface. The Open and Save buttons provide a mechanism for saving and loading selected options—you can save selected options in a file that can be reloaded at any time (a real time saver).

This program demonstrates a common theme you will encounter when developing Cocoa programs, as well as Cocoa front-ends to command-line tools: designing and constructing a user interface that calls a UNIX command-line tool and displays the result of the operation. In order to use CocoaWGet, you will need

to install `wget`. You can do this many ways, but the simplest is to use the Fink installation tool called `dselect`. See the Fink site for more information about the Fink project and installing programs under Fink (http://fink.sourceforge.net/index.php).

NOTE The Fink project simplifies the task of installing Unix open source software on Darwin and Mac OS X. The project maintains a collection of ports, or *packages*, of UNIX programs that users download through a package management tool called `dselect` (like the Debian project's tool of the same name). This tool installs the software on your computer.

Cocoa provides developers with a solid set of frameworks for developing Mac OS X applications. Using these frameworks coupled with Objective-C enables you to create useful applications with sophisticated user interfaces. This chapter takes you through the steps of creating a Cocoa program, from building the interface and creating the classes and instances in Interface Builder, to implementing the code in Project Builder. In addition, I discuss some design user interface issues.

As you proceed through these pages, you will see how simple and intuitive it is to create a Cocoa program. One of the most important aspects of this chapter is showing how simple it is to call UNIX command-line tools from a Cocoa program. You will use this technique repeatedly in future programs for connecting Cocoa interfaces to command-line tools.

6.3 Program requirements

The first step in writing any program is to describe what the program will do. I have found that unless I constrain the problem in the form of a simple textural or graphic description, including requirements and design issues, I tend to develop lots of unnecessary code and add features that are not necessary.

Here is a simple description of the program:

> The CocoaWGet program facilitates the retrieval of files from web and FTP sites using the GNU `wget` program. CocoaWGet provides a GUI front-end for selecting program options and fully supports all the `wget` command-line options. Users using CocoaWGet are not limited in any way and will be able to do anything they can do with the command-line version.
>
> The CocoaWGet program performs the following tasks:
>
> - Lets the user select `wget` options through a GUI interface

- Enables the user to retrieve files from a remote site using the selected options
- Lets the user save the current options to a file
- Enables the user to reloaded the saved options file

The user interface should be orderly and intuitive, and should follow the guidelines outlined in Inside Mac OS X: Aqua Human Interface Guidelines.

6.4 Program design

CocoaWGet is based on the Model-View-Controller (MVC) design pattern, which is a very useful and commonly used design technique for constructing GUI programs. As you learned in chapter 5, MVC is composed of three parts: the model, which holds the program's data state; the view, which displays one or more views of the data; and the controller, which mediates between model and view. This design provides a clear separation of responsibility between program components and encourages reusability, mainly in the model and view components.

Figure 6.2 shows an overview of the MVC pattern applied to the CocoaWGet program.

The program's data resides in the model (`WgetParameters`) and is implemented with `NSMutableDictionary`, a hash table. The key is the command-line parameter

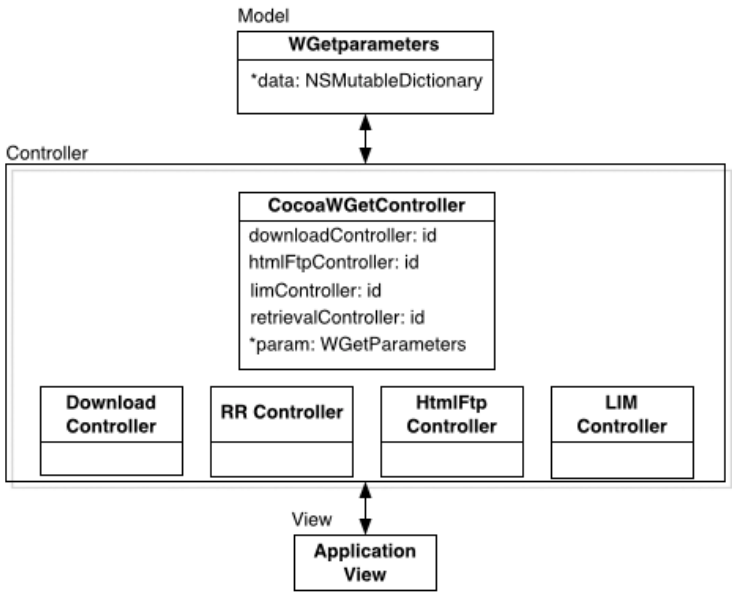

Figure 6.2
Cocoa programs are commonly based on the Model-View-Controller (MVC) design pattern, as shown here applied to CocoaWGet.

and the value is its accompanying value. For example, the `--output-file=[file]` option is used to direct all log messages to a specified file. In this design, `--output-file` is the key, and the file's name is the value.

CocoaWGet has one main controller and four subcontrollers. Each of the subcontrollers mediates information between the data model and its corresponding tabbed pane. For example, the download controller handles the download pane, the HTML/FTP controller handles the HTML/FTP pane, and so on. The main controller oversees the mediation process and handles information between the model and the view. The view is responsible for visually displaying the state of the model, in this case the selected `wget` parameters. Collectively, these components work together to handle all of the application's operations and services.

6.5 Building the interface

The first step in building any Cocoa application is designing and laying out its user interface in Interface Builder. There are many ways to design a user interface, but because this is a Macintosh program, you want to strive to make it as "Mac-like" as possible. Doing so will ensure that your program maintains the Macintosh look and feel and performs as Macintosh users expect.

A good way to begin is to look at well-designed Mac OS X programs and see how their designers constructed the program's interface. There are many examples to choose from, including programs developed by Apple (such as iTunes and Mail) and programs developed by third-party developers (like BBEdit from Bare Bones Software [http://www.barebones.com] and the programs written by the Omni Group [http://www.omnigroup.com]).

You should also invest some time reading about good interface design. Many books, articles, and online sites detail and explain the basic principles of user interface design. The "Resources" section at the end of the book provides some recommendations.

6.5.1 Opening the project

Project Builder simplifies the process of creating programs by providing a set of predefined project templates. The project template defines a set of files, resources, and build options that collectively provide basic application functionality. From this base, you add files, resources, and build options specific to your application. CocoaWGet is based on the Cocoa Application template.

The CocoaWGet project is located in the source_code/chapter06/CocoaWGet folder. Locate this directory from the Finder and double-click on CocoaWGet.pbproj to launch Project Builder and load the CocoaWGet project.

6.5.2 The interface components

Building an interface with Interface Builder is an intuitive process that primarily involves drawing the interface, creating classes, and forming connections between interface elements and program objects. The general cycle is to create your application's interface component, such as a window; populate it with interface controls by dragging each control from the Interface Builder palette to the window; create your program's classes, including methods and data members; and connect the appropriate controls to program objects such as outlets (data members) and actions (methods).

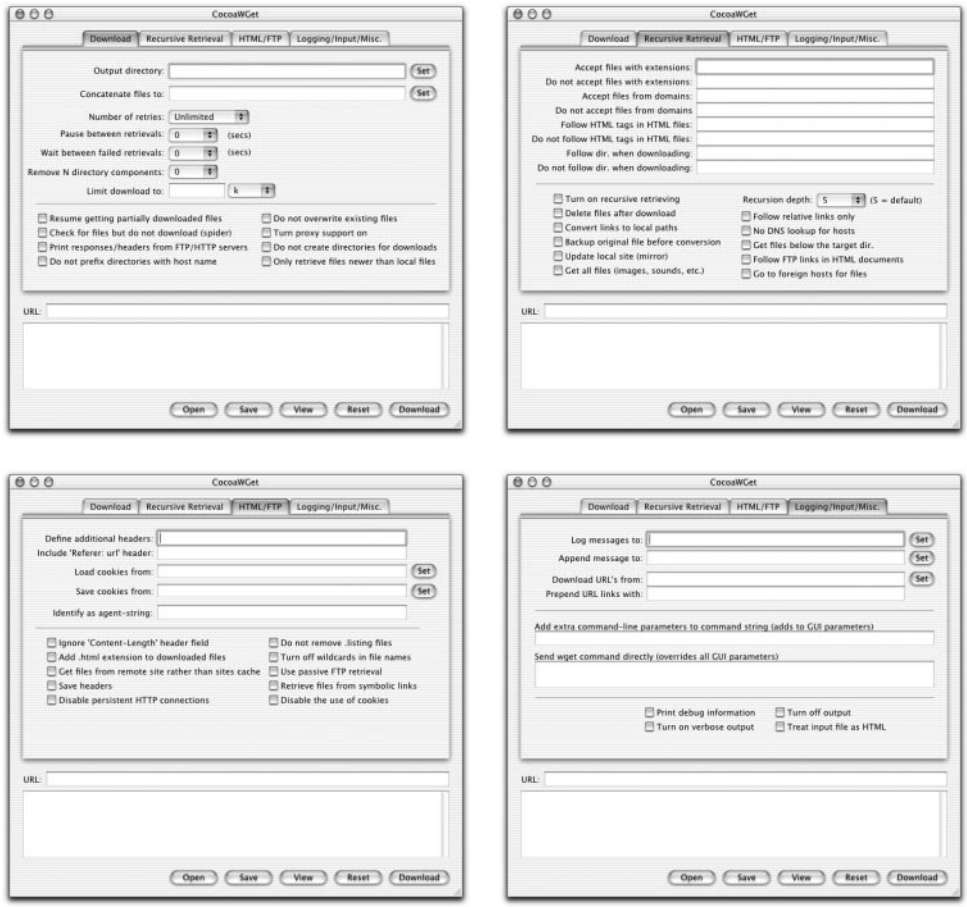

Figure 6.3 The tabbed panes of the CocoaWGet program. Use this figure as a guide for constructing the user interface layout.

Let's begin by looking at the components of the program's user interface within Interface Builder. Open the Resource group in the Groups & Files pane (in Project Builder) and double click on MainMenu.nib; doing so launches Interface Builder and opens the file. If necessary, click on the Instance tab on the MainMenu.nib window and double-click on the Window icon to open the main window.

The CocoaWGet user interface was built by dragging each interface component from the Interface Builder palette to the appropriate location within its tabbed view (see figure 6.3).

Table 6.1 lists the control types for the various interface objects.

Table 6.1 Most types of controls are straightforward. This table clarifies those that may not be obvious.

Pane	Item	Type
Main	Status field at bottom of window	`NSTextField`

Download	• Output directory, Concatenate file to, Limit download to
	• Pop-up menus
	• Checkboxes
		`NSTextField`
		`NSPopupMenu`
		`NSButton`

Recursive Retrieval	• Accept/ Do Not Accept files with extensions, Accept/ Do Not Accept files from domains, Follow/Do Not Follow HTML tags in HTML files, Follow/ Do Not Follow dir. when downloading
	• Recursion depth pop-up menu
	• Checkboxes
		Each is of type `NSForm`
		`NSPopupMenu`
		`NSButton`

HTML/FTP	• Define additional headers
	• Include referer: URL header
	• Load/Save cookies from
	• Identify as agent-string
	• Checkboxes
	• Set buttons
		Each is of type `NSForm`
		`NSTextField`
		`NSButton`
		`NSButton`

Logging/Input/Misc.	• Log/Append Messages To, Download URLs From / Prepend URL Links With
	• Add extra command-line parameters, Send wget command
	• Checkboxes
	• Set buttons
		Each is of type `NSForm`
		`NSTextField`
		`NSButton`
		`NSButton`

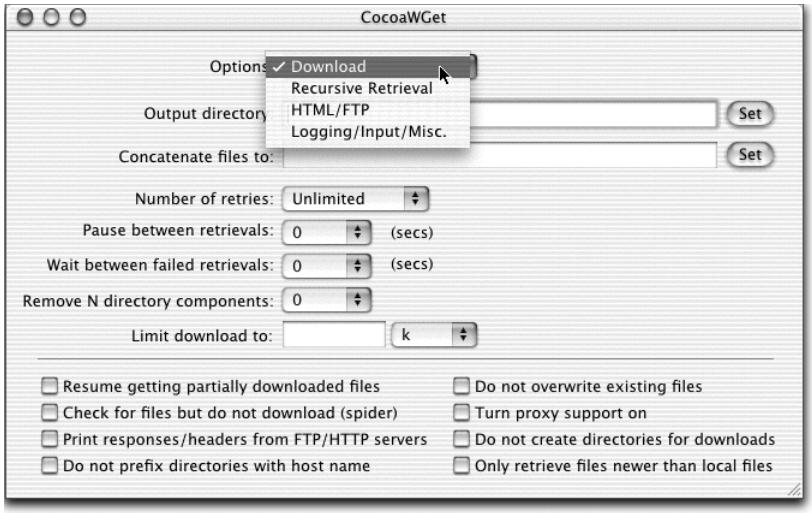

Figure 6.4 An example of one implementation of the CocoaWGet interface

The `wget` program provides the user with many control-line options. One of the challenges in creating a user interface is to logically arrange these options and present them to the user in a clean, orderly way. You can do this by creating a window containing a pop-up menu that lists each category of command-line option (download, logging and input files, http/FTP options, recursive retrieval, and so on; see figure 6.4). When the user selects an option from the menu, the program displays the controls in the window.

Another choice is to create a toolbar at the top (or along the left side) of the window and have each icon represent a different category. When the user selects an icon, the program displays the appropriate controls in the window. Finally, the program can display the options using a set of tab controls.

Each option has its strengths, weaknesses, and design tradeoffs. For this program, you will use the final design choice: tabbed controls, each of which displays a different set of options. Tab controls are a simple, orderly way to present information to the user. Each tab control holds a set of `wget` options, permitting the user to easily navigate between option classes.

6.5.3 Control alignment and spacing

As you build your interface, you should try to adhere to Apple's interface guideline recommendations. Doing so will ensure that your program maintains the look of a Macintosh program. To support the proper placement of controls, Interface

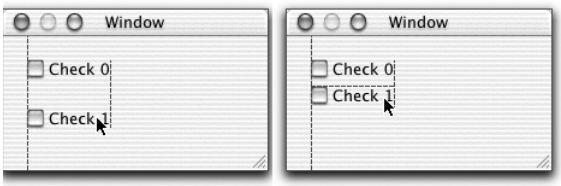

Figure 6.5
Aqua guides help you align controls according to the Apple interface guidelines. The first window shows the display before you reach the correct vertical position; the second window shows the display after positioning.

Figure 6.6
Layout rectangles provide more alignment information about controls.

Builder provides on-screen help for aligning interface elements and controlling the spacing between controls. Let's take a look at some of these features:

- *Aqua guides*—As you drag a control within a window, you will see blue lines appear in the window. These Aqua guides help you line up interface components according to the interface guidelines (see figure 6.5). Using these guides, you can get a quick indication of where to place a control in relation to its window, view, or other controls. For example, the interface guidelines say that there should be 8 pixels between stacked checkboxes. As you stack checkboxes, the Aqua guides steer you to the correct position.
- *Layout rectangles*—To get more visual information about the control layout, use the View Layout Rectangles feature (Command-L). This feature draws a red line around each control, showing more precisely its position in the window (see figure 6.6). Layout rectangles are useful for tasks like consistently aligning text field captions with their corresponding text fields.

Aqua guides and layout rectangles provide you with general visual feedback about the position of controls within a window or dialog. However, sometimes you need to know the relationship between components in exact pixels. For example, to display the number of pixels from a control to the edges of the window, select the control, move the mouse over an empty part of the window, and hold down the Option key.

To display the distance in pixels from the selected control to another control in the window, select a control, move the mouse to the other control, and hold down the Option key (figure 6.7).

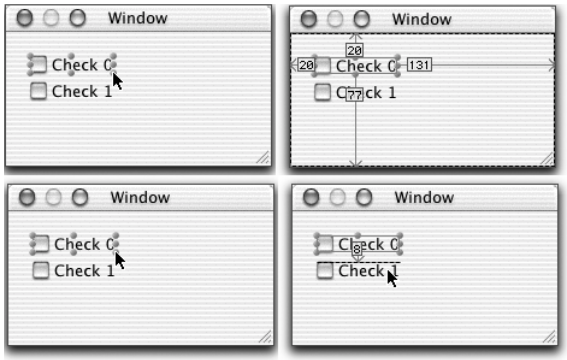

Figure 6.7
The top two screens show an example of displaying the number of pixels from a control to the edges of the window. The bottom two screens display the distance in pixels from the selected control to another control in the window.

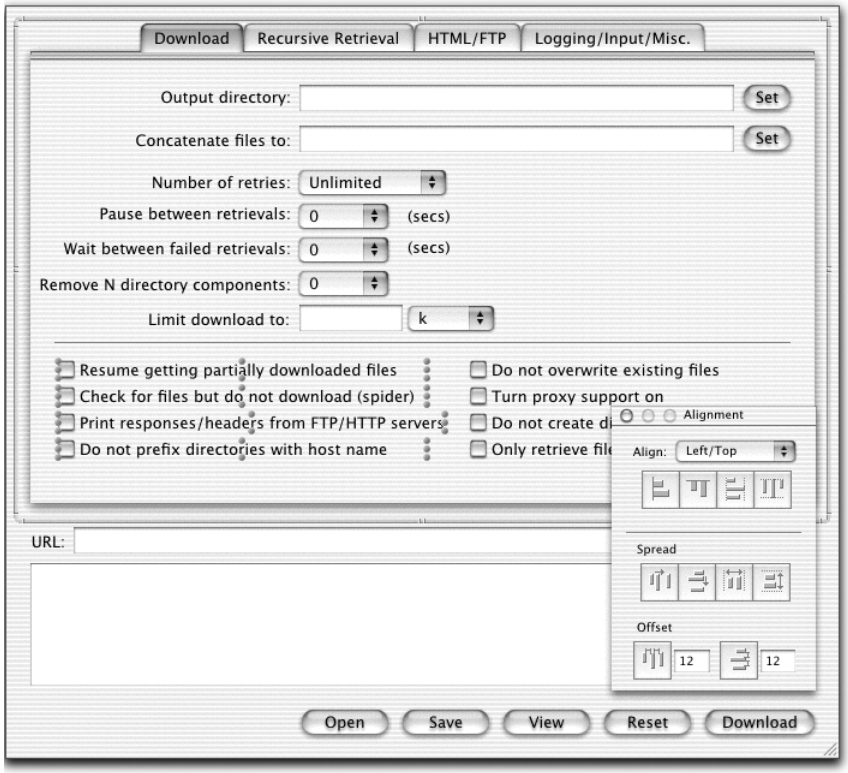

Figure 6.8 The Layout Alignment palette permits you to enter exact values for control alignment.

Another way to align controls is by selecting the Layout→Alignment palette. As you select different controls, the values of the Alignment panel change, enabling you to set specific values for each alignment option (see figure 6.8).

Table 6.2 lists the ways you can align controls within Interface Builder.

Table 6.2 Interface Builder supports several methods for checking the layout of controls within a window.

Name	Command
Aqua guides	Automatic as you drag interface components around a window
View Layout Rectangles	Command-L
View the number of pixels from a control to the edges of the window	Select the control, move the mouse over an empty part of the window, and hold down the Option key
View the distance in pixels from the selected control to another control in the window	Select a control, move the mouse to the other control, and hold down the Option key
Alignment tool	Layout⟶Alignment

As you are constructing the CocoaWGet interface, use these techniques to ensure that you are laying out the window controls correctly.

6.5.4 Forms

You probably noticed the use of forms in some of the dialog boxes. Forms are often used to group related interface components. It is arguable whether forms are the best choice for some aspects of this application, but I've included them as examples of using forms in your interface.

Now that the interface is constructed, it's time to look at how you create the application classes and instances.

6.5.5 Classes and instances

Creating the classes and instances for an application follows a few basic steps. For example, you'll follow these steps to create the classes and instances for the CocoaWGet application. For each tabbed control (four in all), do the following:

1. Click the Classes tab and select `NSObject` (leftmost view).
2. Select Classes→Subclass `NSObject`, or press the Return key.
3. Type the name of each class: `DownloadController` for the Download tab, `RRController` for the Recursive Retrieval tab, `HtmlFtpController` for the HTML/FTP tab, `LIMController` for the Logging/Input/Misc. tab, and `CocoaWGetController` for the main application controller.
4. Select the controller class you just created from the class list (select Tools→Show Info if necessary) and select Attributes from the pop-up menu.

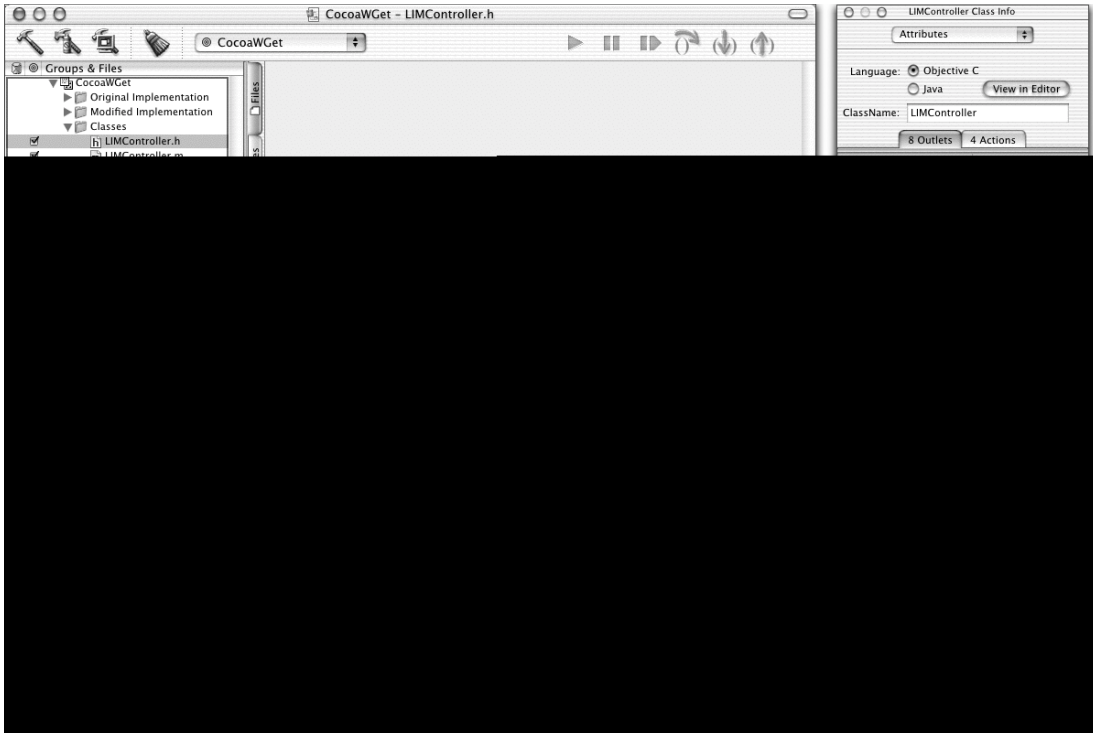

Figure 6.9 An example of how to create outlets and actions for each class

5 Enter the outlets and actions. See the header files in the original project as an example (see figure 6.9).

Outlets and actions

Let's look at outlets and actions in more detail. Connecting outlets and actions is one of the techniques you will use a lot when developing Cocoa programs. Most of us are used to forming these connections programmatically as we develop code; in Cocoa programming, you also do this through Interface Builder. This process can be summarized as follows:

- *Outlet*—An instance variable pointer that stores an interface object's data. In order for the application to exchange data between the model and the view, you need to define outlets: one for each interface component. You connect an outlet by Control-dragging *from* the class instance *to* its corresponding interface element, selecting the outlet's name from the connections list and clicking the Connect button (figure 6.10). The controller uses the outlets to access values in the interface and update the state of the model.

Setting up messaging

In addition to connecting outlets and actions, you can also set up message connections between classes. In the CocoaWGet program, the main controller (`CocoaWGetController`) needs to access the subcontrollers. To accomplish this, Control-drag from the instance that sends the message to the instance that receives the message, select the receiver from the connections list, and click the Connect button (see figure 6.12).

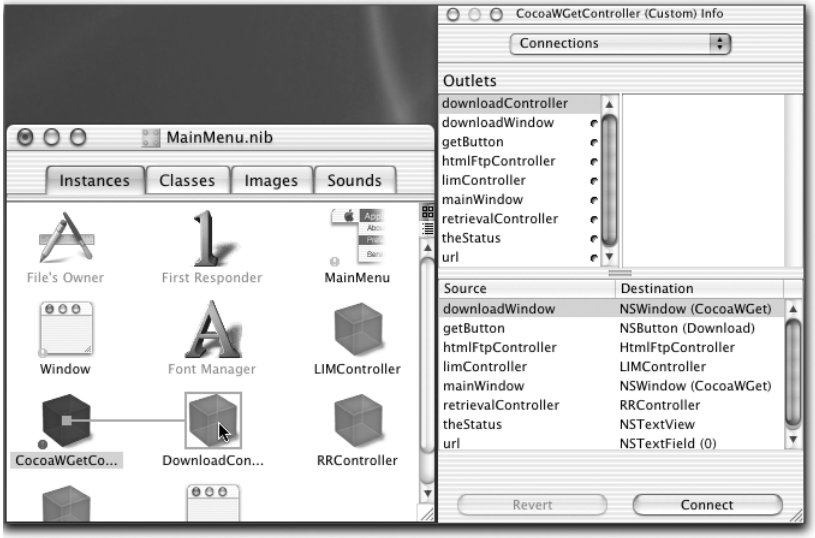

Figure 6.12 To connect messaging between classes, Control-drag from the instance that sends the message to the instance that receives the message, select the outlet or action name from the connections list, and click the Connect button.

Generating interface and implementation files

The last step is to generate the class's interface (.h) and implementation (.m) files and add them to the Project Builder CocoaWGet project. The generated interface file contains the outlets and actions you created, and the implementation file contains the method definitions. In addition, the application's Nib files contain all the application resources, object instances, and connections for the outlets and actions.

You generate the CocoaWGet class files as follows:

1. Click the Classes tab and select a class (`DownloadController`, `RRController`, and so on).

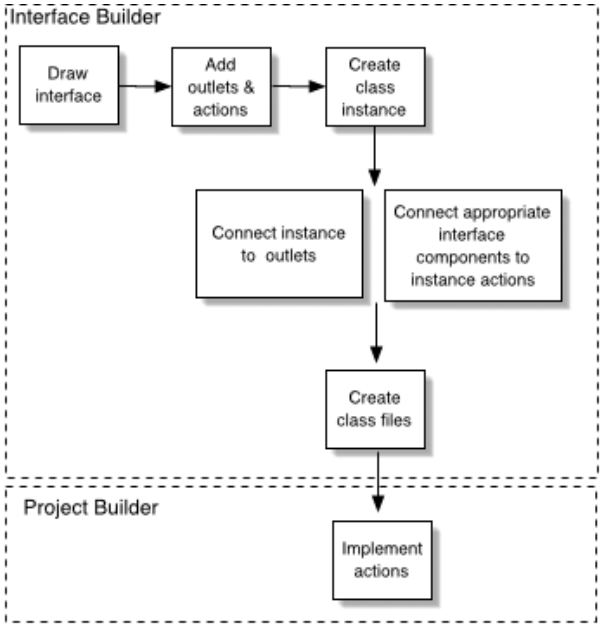

Figure 6.13
The main steps in designing a Cocoa program, from Interface Builder to Project Builder

2 Select Classes→Create Files For [*Class Name*] and click the Choose button in the Save File dialog box to select the directory to save the files in. Interface Builder creates the class files and automatically adds them to the project.

The next section will show you how to add code to the implementation files to extend the behavior you defined in Interface Builder. At this point, you have seen how to build the program's user interface and how to connect interface components to outlets and actions. Figure 6.13 illustrates the steps of building a Cocoa interface.

6.6 CocoaWGet: implementing code with Project Builder

Now that you have the program's interface built, it is time to look at some code. The CocoaWGet program is composed of five controller classes and some support classes. The controller classes include one main application controller that drives the application and four subcontrollers that mediate messages between the main controller and the tabbed view panes. The support classes include a task class that is responsible for running the `wget` UNIX command-line program, collecting the output of the program, and returning the output to the client; the application support class, which handles miscellaneous support tasks for the program; and the parameters class, which stores `wget` options. Figure 6.2 shows a simple class diagram of the program.

Rather than stepping you through the stages of adding code application code to the program, I'll instead detail each class, showing how it works and how it fits into the application. Refer to the code examples here as well as the full source code from the project.

6.6.1 The model

As you recall from the previous discussion of the MVC pattern, the model is responsible for maintaining the data state of the program and responding to client messages that query the state for values or update the state of the model.

WGetParameters class

The CocoaWGet program's model is represented by the `WGetParameters` class (see figure 6.14):

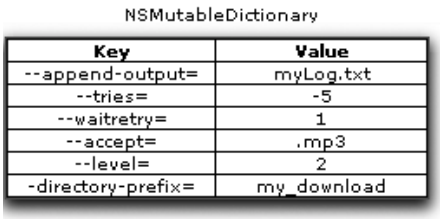

Figure 6.14
The `WGetParameters` class holds data (the model) using an `NSMutableDictionary`, which stores objects in key/value pairs.

```
@interface WGetParameters : NSObject {
  NSMutableDictionary *data;
  NSMutableString *cmdLine;
}
- (NSString *)getValue:(NSString *)key;
- (void)setValue:(NSString *)key: (NSString *)value;
- (void)printData;
- (NSMutableArray *)getData;
- (NSMutableString *)getCommandLine;
- (void)formatCommandLine;
- (void)saveData:(NSString *)fname;
- (void)loadData:(NSString *)fname;
- (void)initToDefaults;
@end
```

The class contains two data members: one holds the data state of the program (the model) implemented as an `NSMutableDictionary`, and the other holds the command line. `NSMutableDictionary` (*mutable* meaning capable or subject to change) is part of the Cocoa Foundation collection classes. It holds objects as key/value associations, similar to a map in the C++ Standard Template Library (STL) or a hash in Perl. For each unique key, there is an associated value. The `WGetParameters`

class uses the dictionary to store each command-line parameter and, if necessary, its corresponding value.

The `WGetParameters` class initializes its data members through its `init` method. The `init` method, like a C++ constructor, is called when the class is instantiated by the runtime system and is typically used for initializing the class's data members. Rather than setting the hash table values within the `init` method, you send a message to `initToDefaults` and have it set the values to their defaults. You do this so the program can reuse the method to respond to the user selecting the Reset button, which also sets the model to its default values. In addition to the dictionary, the class contains an `NSMutableString` data member, which holds the command-line version of the current state of the model.

Let's look at the most important class methods:

- *`getValue` and `setValue`*—Enable controlled access to the class and therefore the model. The `getValue` method takes a key parameter that it uses to look up and return the associated value in the model. The `setValue` method takes two parameters (a key and value) that the class uses to set the value for the associated key.
- *`getData`*—Responsible for taking the current model state and returning an array of each *set* key/value pair. By *set*, I mean a parameter selected by the user in the interface. For example, when the program starts, the data model is set to default values. When the user clicks the Download button, the program controllers query each view, update the model based on the state of the view, and send a `getData` message to the model, which it responds to by returning the selected command-line parameters. The `formatCommandLine` method uses the `getData` method to get the current parameters and convert them to a `wget` command-line representation.
- *`saveData` and `loadData`*—Handle saving the current model to a file, loading a saved file, and populating the model with the stored values. These are two of the more interesting methods. The program uses them to handle this feature. The scenario feature enables the user to save the current setting to a file they can load later. As you can see from the following snippet, the `saveData` method is only one line—this is all it takes to save the contents of an `NSMutableDictionary` to a file:

```
- (void)saveData:(NSString *)fname
{
  [data writeToFile:fname atomically:YES];
}

- (void)loadData:(NSString *)fname
```

```
{
  NSString *s = fname;
  s = [s stringByExpandingTildeInPath];
  [s retain];
  [data release];
  data = [[NSMutableDictionary alloc] initWithContentsOfFile:s];
  if (data == nil) {
    data = [[NSMutableDictionary alloc] init];
    [self initToDefaults];
  }
}
```

The first parameter is the name of the file. The second is a Boolean flag that tells the method how to save the data. If it is `YES`, the method saves the data to a temporary file and, if successful, copies that file over the named file. If `NO`, the method writes the data directly to the specified file without the temporary copy. (Temporary copies protect the user if the power is cut while the file is being written.) The format of the files is XML. Here's an edited example of the XML output:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/
  PropertyList.dtd">
<plist version="0.9">
<dict>
  <key>--accept=</key>
  <string></string>
  <key>--append-output=</key>
  <string></string>
  <key>--backup-converted</key>
  <string>0</string>
  <key>--base=</key>
  <string></string>
  <key>extra-commands</key>
  <string></string>
  <key>raw-command</key>
  <string></string>
</string>
</dict>
</plist>
```

- *`loadData`*—Releases the current model and loads the data from the specified file into a new model. If there is an error (`data == nil`), the method sets the model to its default values.
- *`stringByExpandingTildeInPath`*—Expands a path name that contains ~ (the user's home directory) to a full path name.

Collectively, these methods show how easy it is to serialize data to and from disk.

6.6.2 The view

In the MVC paradigm, the view is responsible for displaying data to the user. When you're creating CocoaWGet's user interface within Interface Builder, you effectively created the application view. Cocoa's Application Kit handles most of the displaying and updating of the view for you.

6.6.3 The controller

The controller is responsible for mediating interaction between the application's model and view. In the CocoaWGet program, one main controller and four subcontrollers handle this aspect of the pattern.

CocoaWGetController class

The `CocoaWGetController` class is the main application controller. It is responsible for routing messages to each of the subcontrollers and handling user interaction with the main application. The application supports actions such as saving and opening the current parameters, resetting the interface and model, and invoking a `wget` retrieval. Here is the interface of the `CocoaWGetController` class:

```
@interface CocoaWGetController : NSObject
{
  IBOutlet id downloadController;
  IBOutlet id htmlFtpController;
  IBOutlet id limController;
  IBOutlet id retrievalController;
  IBOutlet NSTextView *theStatus;
  IBOutlet NSTextField *url;
  IBOutlet NSWindow *mainWindow;
  IBOutlet NSWindow *downloadWindow;

  NSString *directory;
  WGetParameters *param;
}

- (IBAction)handleDownload:(id)sender;
- (IBAction)handleOpen:(id)sender;
- (IBAction)handleReset:(id)sender;
- (IBAction)handleSave:(id)sender;
- (IBAction)handleViewParams:(id)sender;

- (void)reset:(id)sender;
- (void)raiseSheet;
- (void)closeSheet:(id)sender;
- (void)displayCmdLine:(NSString *)headerStr;
@end
```

The class contains several data members, which correspond to the subcontrollers, interface elements, and model. The subcontrollers' (`downloadController`, `htmlFtpController`, `limController`, and `retrievalController`) data members enable `CocoaWGetController` to access each of the subcontrollers. You set the connection between `CocoaWGetController` and these controllers in Interface Builder by Control-dragging from the `CocoaWGetController` instance to each subcontroller instance and selecting the corresponding outlet. With these connections intact, the `CocoaWGetController` can talk to any of the subcontrollers.

The `CocoaWGetController` class uses the next data members, `theStatus` and `url`, to access interface elements—in this case, the status and URL fields. The status field holds the output messages from the `wget` program, as well as any status information messages inserted by the CocoaWGet program. The URL field contains the source URL.

After the user selects options and clicks the Download button, CocoaWGet begins the download process. At this point, the program should inform the user of its operations so the user knows what is going on. This is somewhat different from the way many UNIX programs work. Typically, a UNIX program remains silent, showing messages only when a warning or an error occurs. The idea behind this design choice is that users only need to worry if they see output. Most GUI interfaces instead display the status of the operation to inform the user that the program is functioning and processing their request. CocoaWGet displays a dialog called a *Sheet* during the download process. (As you'll recall from chapter 1, Sheets are modal dialog boxes. When an application displays a Sheet, it appears attached to an application's document or window.)

Sheets are new to Mac OS X. In order for the program to display the Sheet in the correct window, you need to keep a pointer to the window to which the Sheet is attached. To accomplish this, you use the `mainWindow` data member. This member holds a pointer to the main application window, which is set in Interface Builder by Control-dragging from the `CocoaWGetController` to the main application window and setting the connection to the `mainWindow` outlet. This member is used as a parameter to `NSApp`'s `beginSheet` method. The `downloadWindow` data member holds a pointer to the window that the Sheet uses to display the download message. You create this window and form the connection between it and the `downloadWindow` data member in Interface Builder.

Along with these data members, the class also holds a pointer to the program's home directory, set in the `init` method, which it uses as the default location to store downloaded files and the application's model class (`param`). The class uses the model object to access the application model.

In addition to the data members, `CocoaWGetController` implements several methods that enable it to respond to user requests or actions, display application information, and interact with the model. The `handle` family of methods responds to user requests:

- *`handleSave`*—Responds to messages to save the currently selected parameters to a file. Each of these methods uses support methods defined in the `AppSupport` class, discussed later in the section.
- *`handleReset`*—Sets the model to its default values and sends a message to the view to update its display.
- *`handleViewParams`*—Displays the currently selected parameters in the status field as a `wget` command line.
- *`handleDownload`*—The most interesting of the defined methods. It is in charge of collecting and formatting `wget` parameters and running the `wget` task to retrieve all files based on user selections:

```
- (IBAction)handleDownload:(id)sender
{
  NSMutableArray *args;
  MyTask *task;

  [self displayCmdLine:@"Downloading files, please wait..."];

  [limController getParameters:param];
  [downloadController getParameters:param];
  [retrievalController getParameters:param];
  [htmlFtpController getParameters:param];

  [param setValue:@"url":[url stringValue]];

  [self raiseSheet];
  args = [param getData];

  task = [[MyTask alloc] init];
  [task runTask:WGET_CMD theDirectory:directory
        theArgs:args getOutputFrom:1];

  [self closeSheet:sender];

  [AppSupport setStatusMsgWithDate:theStatus theMsg:[task output]];
  [self displayCmdLine:@"Download complete."];

  if ([task exitStatus] != 0)
    NSRunAlertPanel(@"Error getting files",
    @"wget returned an error.", @"OK",     NULL, NULL );

  [task release];
}
```

The `handleDownload` method works as follows:

1. It prints a message to the status text area telling the user that the download is beginning, and updates the application model by sending a message to each subcontroller to query its controls and update the model to reflect the current settings.
2. It sends a message to the `raiseSheet` method to display the download Sheet, which has the side effect of disabling the interface for user interaction.
3. It sends a message to the model to return the parameters in an array, where each element is a key/value parameter pair.
4. To run the `wget` task, the method instantiates a `MyTask` object and sends a message to `runTask`, passing the launch path to the `wget` program (/sw/bin/wget), the directory to store the retrieved file under, the command-line arguments, and where to read the `wget` output (0 for standard out or 1 for standard error). The `MyTask` class, which does much of the real work of interacting with the UNIX layer, is discussed later in this section.
5. When the `runTask` method finishes, the download is complete. `handleDownload` closes the Sheet and updates the status field and prints the `wget` output.
6. The task object is released.

DownloadController, RRController, HtmlFtpController, and LIMController classes

In addition to the main application controller (`CocoaWGetController`), CocoaWGet uses four subcontrollers that operate under the control of the main controller. As discussed in section 6.5.3, the main program window has four tabbed controls: each pane holds a related group of `wget` parameters and is controlled by a different controller. The Download pane is mediated by an instance of the `DownloadController` class; the Recursive Retrieval pane is mediated by an instance of the `RRController` class; and this pattern continues for the remaining panes and controllers. Each subcontroller implements similar functionality, so let's look at one of the controllers as an example.

The `DownloadController` class implements the following data members and methods:

```
@interface DownloadController : NSObject
{
  IBOutlet NSTextField *concatFilesTo;
  IBOutlet NSTextField *limitDownloadSizeTo;
  IBOutlet NSPopUpButton *limitDownloadSizeType;
  IBOutlet NSButton *noDirCreateOnDownload;
  IBOutlet NSButton *noFilesUnlessNewerThanLocal;
```

```
  IBOutlet NSButton *noHostnamePrefix;
  IBOutlet NSPopUpButton *nRetries;
  IBOutlet NSTextField *outputDir;
  IBOutlet NSButton *overwriteFiles;
  IBOutlet NSPopUpButton *pauseBetweenRetrievals;
  IBOutlet NSButton *printServerResponse;
  IBOutlet NSButton *proxyOn;
  IBOutlet NSPopUpButton *removeNDirComponent;
  IBOutlet NSButton *resumeDownload;
  IBOutlet NSButton *spider;
  IBOutlet NSPopUpButton *waitBetweenFailedRetrievals;
}
- (IBAction)handleSetConcatTo:(id)sender;
- (IBAction)handleSetOutputDir:(id)sender;
- (IBAction)reset:(WGetParameters *)param;
- (void)getParameters:(WGetParameters *)param;
@end
```

The data members perform functions similar to the other classes we looked at; primarily, they provide an access point to get the state of the pane's interface controls.

You can divide the methods into two categories: methods that react to messages sent by the instance in response to user interface selections, and methods that interact with the data model. The `handleSetConcatTo` and `handleSetOutputDir` methods deal with user selections. For example, `handleSetOutputDir` is responsible for responding when the user clicks the Set button and getting a directory from the user. The method uses the `getDirectory` methods, which you will implement in the `AppSupport` class (discussed later in the section):

```
- (IBAction)handleSetOutputDir:(id)sender
{
  [outputDir setStringValue:[AppSupport
                             getDirectory:@"Output Directory"]];
}
```

The `reset` method handles updating the interface (view) to reflect the current state of the model. For each interface component, you get the corresponding value from the model and send it as a parameter to the control's `set` method:[1]

```
- (IBAction)reset:(WGetParameters *)param
{
  [limitDownloadSizeTo setStringValue:[param getValue:@"--quota="]];
  [outputDir setStringValue:[param getValue:
   @"--directory-prefix="]];
  [pauseBetweenRetrievals setStringValue:[param getValue:
   @"--wait="]];
```

[1] The naming scheme of `-something=` is used to map keys to `wget` command-line options.

```
    [removeNDirComponent setStringValue:[param getValue:
     @"--cut-dirs="]];
    [concatFilesTo setStringValue:[param getValue:
     @"--output-document="]];

    if ([[param getValue:@"--timestamp"] isEqualToString:@"1"])
      [noFilesUnlessNewerThanLocal setState:NSOnState];
    else
      [noFilesUnlessNewerThanLocal setState:NSOffState];
    // …
}
```

Let's look at a few of these statements. To set the value of the Quota text field, you first get the value in the model for the key `--quota=` and pass it as a parameter to `setStringValue`. To set the state of a checkbox (`NSButton`), you determine the key's value in the model. If it equals 1, the box should be checked, so you send it a `setState` message with `NSOnState` as its parameter. Conversely, if the box should be unchecked, you send a `setState` message with `NSOffState` as its parameter. You repeat this process for each control on the pane.

The `getParameters` method gets the current user settings from the view and updates the model according to these choices:

```
- (void)getParameters:(WGetParameters *)param
{
  NSString *s;

  [param setValue:@"--quota=":[limitDownloadSizeTo stringValue]];
  [param setValue:@"Q-size":[limitDownloadSizeType
   titleOfSelectedItem]];
  [param setValue:@"--tries=":[nRetries titleOfSelectedItem]];
  [param setValue:@"--output-document=":[concatFilesTo
   stringValue]];
// …
}
```

To set a model's value, you first get the current value of the control and send a message to the model, passing the options key as the first parameter and the retrieved value as the value parameter. You repeat this process for each control on the pane.

Collectively, these methods, as well as the similar methods in the other subcontroller classes, work to mediate information between the application's view and data model, and are managed by `CocoaWGetController`.

AppSupport support class

CocoaWGet uses two additional classes that provide support functions for the program: `AppSupport` and `MyTask`. The `AppSupport` class, as its name suggests, provides basic support for the program:

```
@interface AppSupport : NSObject {
}
+ (NSString *) getFilename:(NSString *)title;
+ (NSString *) getDirectory:(NSString *)title;
+ (void)setStatusMsgWithDate:(NSTextView *)statusField
  theMsg:(NSString *)msg;
+ (NSString *)getSaveFile:(NSString *)title;
+ (void)scrollStatus:(NSTextView *)statusField;
 @end
```

The `AppSupport` class only contains *static methods*. Static methods are preceded with a `+`, as opposed to the `-` character (which indicates an instance method). You use a static method through its class rather than its instance variable, so you can use such methods without creating an instance of the class. This technique is useful in some contexts where you want to provide functionality but do not need to maintain class state. The following example demonstrates the syntax for an instance method and a factory method:

```
// Instance method
- (void)foo;
// Factory method
+ (void)foo;
```

The `getFilename`, `getDirectory`, and `getSaveFile` methods prompt the user for filenames the program uses in various tabbed panes. Each method takes one parameter: the title of the dialog. The first two methods (`getFilename` and `getDirectory`) use the Application Kit class `NSOpenPanel` (specifically, the `openPanel` method), which prompts the user for the name of a file to open. The `getSaveFile` method uses `NSSavePanel`'s `savePanel` method. Here's the `AppSupport` class's `getFilename` method:

```
+ (NSString *) getFilename:(NSString *)title
{
  NSString *s = @"";
  NSOpenPanel *panel;
  int result;
  panel = [NSOpenPanel openPanel];
  [panel setCanChooseFiles:TRUE];
  [panel setCanChooseDirectories:FALSE];
  [panel setAllowsMultipleSelection:FALSE];
  [panel setTitle:title];
  result = [panel runModalForDirectory:NSHomeDirectory()
```

```
            file:nil types:nil];
    if(result == NSOKButton) {
      NSArray *retArray = [panel filenames];
      s = [NSString stringWithFormat:@"%@",
          [retArray objectAtIndex:0]];
    }
    return s;
  }
```

By changing the parameters of the `openPanel` method (bold in the listing), you can alter the behavior of the displayed dialog box. For example, `getFilename` only needs a single filename from the user, so you set `setCanChooseFiles` to `TRUE` and `setCanChooseDirectories` and `setAllowsMultipleSelection` to `FALSE`. The `getDirectory` method prompts the user for a directory name, so you set `setCanChooseFiles` and `setAllowsMultipleSelection` to `FALSE`, and `setCanChooseDirectories` to `TRUE`. Both methods return either the file or directory name as an `NSString`.

MyTask support class

The `MyTask` class is one of the more interesting classes in the project. This class is responsible for running a task (program), collecting the results of the run, and returning the result to the user. Is uses the Foundation class `NSTask` class to do so.

The `NSTask` class facilitates running a program as a subprocess of the active program, as well as monitoring and interacting with the execution of the subprocess. In a sense, this is similar to the UNIX `fork/exec` model of running a child process of a parent. With `NSTask`, there are two ways to run subprocess: you can run the process in the environment it inherits from its creator process or use the `NSTask launch` method. The following example demonstrates how to use the first method in Objective-C:

```
NSTask *task = [NSTask launchedTaskWithLaunchPath:path
                arguments:argumentArray];
NSLog("task returned: %@", [task terminationStatus];
```

You launch a subtask using the `launchedTaskWithLaunchPath` method, which takes two arguments: the absolute path to the process you wish to run and any arguments you wish to pass to the process. For example, to use this call to run `wget`, the path parameter would hold the absolute path to the `wget` program, and the argument parameter would hold any `wget` command-line arguments. Note that the subprocess inherits its runtime environment from the calling process. In addition, `launchedTaskWithLaunchPath` is a static method, so there is no need to instantiate the `NSTask` class. When the call returns, you can use the returned `NSTask` object to interact with the task.

This is a simple and straightforward method of running a subprocess, but it does not work when you need to alter the runtime environment of the subprocess. For example, imagine that you wish to capture the output of the subprocess. In this case, the `launchedTaskWithLaunchPath` method will not work.

If you instead run a subprocess using the `NSTask launch` method, the `launch` method, coupled with supporting `NSTask` methods, provides more control over the launching of the subprocess. Table 6.3 lists the methods you use to alter the subtask's runtime environment.

Table 6.3 You use these methods to set the runtime environment under which a task executes.

Name	Description
setCurrentDirectoryPath:path	Sets the current directory path of the subtask environment to path
setStandardOutput:arg	Sets standard output as the receiver of arg, which is an NSFileHandle or NSPipe Means that information sent to standard output now goes to arg
setStandardInput:arg	Sets standard input for the receiver to arg, which is an NSFileHandle or NSPipe object
setStandardError:arg	Sets standard error for the receiver of to arg, which is an NSFileHandle or NSPipe object
setLaunchPath:path	Sets the launch path—the path to the program to execute—to path
setArguments	Sets the command-line arguments to arg
launch	Launches the subprocess based on the set parameters

Let's look at the implementation of `MyTask`'s `runTask` method:

```
- (void)runTask:(NSString *)taskName
   theDirectory:(NSString *)dir
       theArgs:(NSMutableArray *)args
       getOutputFrom:(int)outType;
{
  NSPipe *pipe = [NSPipe pipe];
  NSFileHandle *readHandle = [pipe fileHandleForReading];
  NSData *inData = nil;

  m_taskName = taskName;
  m_directory = dir;
  m_args = args;

  task = [[NSTask alloc] init];

  [task setCurrentDirectoryPath:dir];

  if (outType == 0)
```

```
    [task setStandardOutput:pipe];
  else
    [task setStandardError:pipe];
  [task setLaunchPath:taskName];
  [task setArguments:args];
  [task launch];
   while ((inData = [readHandle availableData])
     && [inData length]) {
     NSString *s = [[NSString alloc] initWithData:inData
                  encoding:NSASCIIStringEncoding];
    [m_taskOutput appendString:[NSString stringWithFormat:
    @"%@ ", s]];
    [s release];
  }
  [task release];
  task = nil;
}
```

The method first creates instances of `NSPipe` and `NSFileHandle`. The `NSFileHandle` instance will read the piped data sent from the subtask. Next, the method creates an instance of the `NSTask` object and sets the task's environment through the various set calls (see table 6.3). The `setStandardOutput` and `setStandardError` methods enable you to set up a pipe between the parent and subprocess. Depending on the passed parameter, you send a message to the task object telling it to attach one pipe end-point to standard output or standard error. For example, if the caller passes `0` as the `outType` parameter, the pipe is set to standard output: all messages written to standard output will go to the pipe.

Next, the subtask is run using the `launch` method. The `while` loop reads each message `wget` writes into the `inData` variable, converts it to an `NSString`, and appends it to `m_taskOutput`. The `inData` data member is of type `NSData`, which is a wrapper for a sequence of bytes. Once the method completes, the output of the subtask is stored in `m_taskOutput`, which the client code accesses by sending an `output` message to the object.

6.7 Program extensions

The program is most of the way there, but it is not finished. One of the biggest omissions is the fact that the user has no way to cancel file retrieval. You also need to add an application icon and help files. In the current version of the program, the location of the `wget` program is hard-coded to /sw/bin/wget in CocoaWGetController.m. Another logical addition would be to permit the user to select the default location of the `wget` program; ideally, you could place this (and perhaps other options) in a Preference dialog. Let's look at a few finishing touches.

6.7.1 Letting the user cancel downloads

Ideally, instead of displaying a Sheet with a static message, the program should enable the user to stop a download in progress. This functionality is important from a usability standpoint, as well as a Macintosh design point of view. As I've pointed out, Macintosh programs should put the user in the driver's seat—the user should have complete control over the interface of the program, including canceling running operations. In the current implementation, a user may choose bad parameters and have no way to cancel a potentially long download (of course, there is always `kill -9 [pid]`[2]).

Addressing this limitation would require some reworking of the `MyTasks` class and the current implementation of `CocoaWGetController`. Rather than build a new implementation from scratch, let's follow the age-old programming paradigm of reusing and extending (stealing!) existing code: you will use sample code from Apple and adapt it to fit your needs. This new code is part of a sample program called Moriarity that is available from Apple's Cocoa sample site (http://developer.apple.com/samplecode/Sample_Code/Cocoa/Moriarity.htm).

The CocoaWGet project already contains an implementation of the new feature. Let's begin by running each implementation a few times to get a feel for their user-level differences. Within the Project Builder's Groups & Files pane are two groups: Original Implementation and Modified Implementation (see figure 6.15).

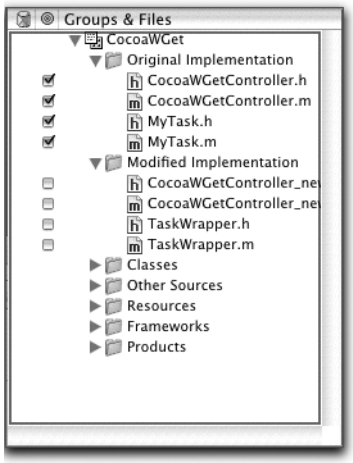

Figure 6.15
The Original Implementation and Modified Implementation groups hold the source files that distinguish the different versions of the CocoaWGet program.

[2] In UNIX, each running process has a unique process identifier, or pid. One way to get the pid of a running process is using the `ps` and `grep` commands: `ps aux | grep [process-name]`.

Each group contains different source files that distinguish the implementations of the program. All other project source files remain the same. The Original Implementation group holds four files that implement the original version of the program. The Modified Implementation group contains the new files that enable the user to interrupt a download.

By the way, this is also a good example of how to use targets within a project (see chapter 3 for more information about targets).

To run the original implementation, open the Original Implementation group by clicking on its disclosure triangle (to the left of the group) and select the checkbox for each file. Next, open the Modified Implementation group and make sure there are no enabled checkboxes. Build and run the program.

To try the other version, deselect the files in the Original Implementation group and select those in the Modified Implementation group. Once again, build and run the program.

The second version provides a much better user experience by letting the user stop a download in progress. In addition, the program displays messages directly to the status field as the program receives them from `wget`.

Now, let's look at the code to get a sense of the differences between the versions. Two changes implement the new additions: modified code in the `CocoaWGetController` class and a new class called `TaskWrapper` that replaces the `MyTask` class. Let's start with `CocoaWGetController`:

```
@interface CocoaWGetController : NSObject <TaskWrapperController>
{
  IBOutlet id downloadController;
  IBOutlet id htmlFtpController;
  IBOutlet id limController;
  IBOutlet id retrievalController;
  IBOutlet NSTextView *theStatus;
  IBOutlet NSTextField *url;
  IBOutlet NSWindow *mainWindow;
  IBOutlet NSWindow *downloadWindow;

  IBOutlet NSButton *getButton;
  NSString *directory;
  WGetParameters *param;
  BOOL retrievalInProgress;
  TaskWrapper *task;
}

- (IBAction)handleDownload:(id)sender;
- (IBAction)handleOpen:(id)sender;
- (IBAction)handleReset:(id)sender;
- (IBAction)handleSave:(id)sender;
- (IBAction)handleViewParams:(id)sender;
```

```
- (void)reset:(id)sender;
- (void)raiseSheet;
- (void)closeSheet:(id)sender;
- (void)displayCmdLine:(NSString *)headerStr;
@end
```

The first change involves adding a protocol list to the class declaration. The protocol list makes the declared methods under the protocol name accessible to the class. For this code to work correctly, you must import the header file that contains the protocol, in this case TaskWrapper.h.

```
#import "TaskWrapper.h"
```

You use the `getButton` data member to access the Download button, enabling the user to initiate a download as well as cancel one. The `retrievalInProgress` data member acts as a flag specifying whether a download is in progress. Because the `wget` program runs asynchronously with the interface, this flag is necessary to indicate the status of the download. This final data member, `task`, points to the task object that runs and manages the `wget` command.

The main differences in the implementation files are a new implementation of the `handleDownload` method and the addition of callback methods:

```
- (IBAction)handleDownload:(id)sender
{
  NSMutableArray *args;

  /*
   Update the model by getting the values from the controls
   and setting the model (param).
   */
  [limController getParameters:param];
  [downloadController getParameters:param];
  [retrievalController getParameters:param];
  [htmlFtpController getParameters:param];

  [param setValue:@"url":[url stringValue]];
  args = [param getData];

  if (retrievalInProgress) {
    // This stops the task and calls our callback (-processFinished)
    [task stopProcess];
    // Release the memory for this wrapper object
    [task release];
    task=nil;
    return;
  }
  else {
    // If the task is still sitting around from the
    // last run, release it
    if (task!=nil)
```

```
      [task release];
    // Let's allocate memory/initialize a new TaskWrapper
    // object, passing in ourselves as the controller for
    // this TaskWrapper object, the path to the command-line
    // tool, and the contents of the text field that
    // displays what the user wants to search on
    task = [[TaskWrapper alloc] initWithController:self
             arguments:[NSArray
             arrayWithObjects:WGET_CMD,@"--help",nil]];
    // kick off the process asynchronously
    [task startProcess:WGET_CMD theDirectory:directory
     theArgs: args];
  }
}
```

Like the original version, it first updates the contents of the model based on interface selections and fills an array with command-line options. Next, it checks the `retrievalInProgress` flag to see if the program is retrieving files. If yes, is stops the retrieval by sending a `stopProcess` message to the task object and releases the task memory. Otherwise, a new task object is created and initialized, and a message is sent to the task to launch the `wget` process.

The other changes to the class involve adding implementations for the `TaskWrapperController` protocol methods. These methods are slightly modified versions of sample code from Apple, changed primarily to fit into our program's design. The source code contains detailed comments from Apple, describing its use and operation. Overall, these methods respond to messages sent from the `TaskWrapper` class initiated by either user events or the invocation or termination of the `wget` process.

The most interesting changes come in the `TaskWrapper`. This class is part of the sample program Moriarity. The class has some interesting features and contains the core functionality you require that enables users to stop a download in progress. The code for the class is well documented, so I will limit my observations to those that affect how it works and interacts with the CocoaWGet program. Let's look at the `startProcess` method:

```
- (void)startProcess:(NSString *)taskName
                     theDirectory:(NSString *)dir
                     theArgs:(NSMutableArray *)args
{
  [controller processStarted];
  task = [[NSTask alloc] init];
  [task setStandardOutput: [NSPipe pipe]];
  [task setStandardError: [task standardOutput]];

  [task setLaunchPath: taskName];
```

```
    [task setArguments: args];
    [task setCurrentDirectoryPath:dir];

    [[NSNotificationCenter defaultCenter] addObserver:self
                  selector:@selector(getData:)
                  name: NSFileHandleReadCompletionNotification
                  object: [[task standardOutput]
                            fileHandleForReading]];

    [[[task standardOutput] fileHandleForReading]
       readInBackgroundAndNotify];
    [task launch];
}
```

The method first sends a `processStarted` message to the main controller (`CocoaWGetController`) to set the `retrievalInProgress` flag to `true`, clear the status field, and change the name of the Download button to Stop:

```
- (void)processStarted
{
  retrievalInProgress = YES;
  [AppSupport setStatusMsgWithDate:theStatus theMsg:@""];
  [getButton setTitle:@"Stop"];
}
```

This indicates to the user that clicking the Stop button will stop the current download. Next, the method sets up the environment as before.

The next step is to register the object with the notification center (`NSNotificationCenter`):

```
- (void)addObserver:(id)anObserver selector:(SEL)aSelector
name:(NSString *)notificationName object:(id)anObject
```

```
[[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(getData:)
        name: NSFileHandleReadCompletionNotification
        object: [[task standardOutput] fileHandleForReading]];
```

According to the `NSNotificationCenter` documentation, the `NSNotificationCenter` object is implemented as a dispatch table. Clients register with the notification center; when a notification occurs, the notification center dispatches the message to the registered objects to handle the request.

In this case, when a notification named `NSFileHandleReadCompletionNotification` (`notificationName`) containing the object of type `TaskWrapper` is posted, [[`task standardOutput`] `fileHandleForReading`]] receives a `getData` message. This message enables asynchronous notification when `wget` writes messages (see the source code for extensive comments from Apple):

```
- (void) getData: (NSNotification *)aNotification
{
  NSData *data = [[aNotification userInfo]
                objectForKey:NSFileHandleNotificationDataItem];
  if ([data length]) {
    [controller appendOutput: [[[NSString alloc] initWithData:data
   encoding:NSUTF8StringEncoding] autorelease]];
  }
  else {
    [self stopProcess];
  }

  [[aNotification object] readInBackgroundAndNotify];
}
```

With these additions, the CocoaWGet program is far more functional and useful to users. All that remains is adding the program icon and help files.

6.7.2 The application icon

Application icons help users identify programs, so it's a good idea to make your icon mnemonic. A good rule of thumb is to design as simple an icon as possible, without clutter and unnecessary components. Mac OS X supports advanced graphics in icons including photo-realistic icons and antialiasing.

Adding an application icon for the program takes a few steps. First, you create your icon using a graphics program such as Photoshop, Graphics Converter (http://lemkesoft.com/us_gcabout.html), or General Image Manipulation Program (GIMP; http://fink.sourceforge.net/pdb/package.php/gimp). Next, use Apple's Icon Composer to create an icon (.icns) file from the saved graphics file. Finally, add the .icns file to the CocoaWGet project.

Creating the icon

For CocoaWGet, I used GIMP to create the icon. Follow these steps:

1. Launch GIMP. Create a new file, setting the width and height to 128 pixels.
2. Set the file type to Transparent to ensure the icon will display correctly in Finder windows and the Dock, rather than having a colored background.
3. Create the icon and save the file in TIF format.

Creating the .icns file

To create the .icns file, follow these steps:

1. Launch Icon Composer, located in /Developer/Application. There are four image sizes, each used for a different display of the icon.

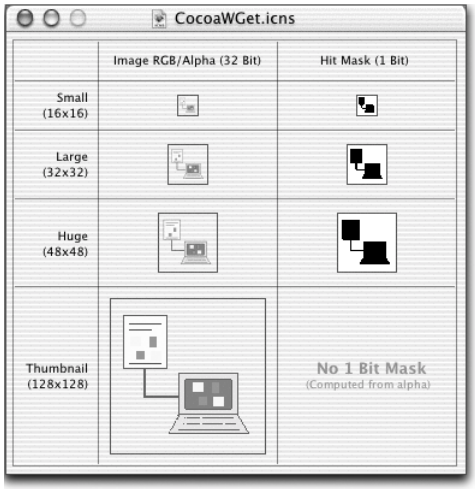

Figure 6.16
Apple's Icon Composer enables you to save icons developed in a graphics program to an .icns file, which Mac OS X programs use to display their application icons.

2. Double-click on the 128x128 box (Thumbnail), select the file that contains your icon (icon_128.tiff), and click the Open button. Icon Composer imports the icon and displays it in the Thumbnail box (see figure 6.16).
3. Repeat this process for each of the other icon sizes, answering Yes if asked to scale the icon.
4. Save this file as CocoaWGet.icns.

Adding the icon to the project

To add the icon to the project, follow these steps:

1. Launch Project Builder and open the CocoaWGet project.
2. Highlight the Resource group, select Project→Add Files, and choose the icon file. You can also add the file by dragging it from the Finder window to the Resource folder in the Groups & Files pane.
3. Select the Targets tab, click on the CocoaWGet target, and click on the Application Settings tab.
4. Scroll to the icon category and enter **CocoaWGet.icns** in the Icon File text field.
5. Perform a make clean (Shift-Command-K) and rebuild the program (Command-B).
6. When you move the program from the CocoaWGet/build folder to the Application folder, you will see your new icon.

6.7.3 The help file

Online help provides users with easily accessible information about program operations and features. Historically, Macintosh applications provide application information through the program's About Box and online help. Under Mac OS X, the About Box displays version and copyright information, as well as the authors of the program, and possible contact information.

The Online Help menu is always the rightmost menu in an application's menu bar. Users of your program use online help to get information about how the program works, help with its features, and possibly pointers to more information. You write help files in HTML and display help files with Apple's Help Viewer, a lightweight browser conforming to the HTML 3.2 standard.

Adding online help to a Cocoa application is a relatively painless process and typically includes the following steps.

1. Create your help files in your favorite HTML editor (compliant with HTML 3.2).
2. Drag the folder that contains the help files onto the Apple Help Indexing Tool icon (/Developer/Application).
3. Add help setting to your application and rebuild the project.

Let's take a look a how to add simple documentation to the CocoaWGet program.

Creating HTML documentation

The first step is to create your documentation in HTML, making sure that it conforms to HTML 3.2. For this task, I use BBEdit, a text-based, non-WYSIWYG editor.

Open the CocoaWGet folder and open the folder called CocoaWGet Help. This folder holds all help files for the program. There are different ways to structure your documentation, but let's keep it as simple as possible.

Under this folder are two subfolders (graphics and html) and a single HTML file called cocoawgettitle.html:

```
<HEAD>
<META NAME="AppleTitle" CONTENT="CocoaWGet Help">
</HEAD>
<TITLE>Test Help</TITLE>
<META NAME="AppleFont" CONTENT="Osaka">
<META NAME="AppleIcon" CONTENT="Test%20Help/icon_16.gif">
<META NAME="AppleSearchResultsFont" CONTENT="Osaka">

<img src="graphics/icon_16.gif" alt="" id="icon" width="16"
height="16" align="left">
<h3>CocoaWGet Help</h3>
```

```
Help is available for the following items

<ul>
  <li><a href="html/cocoawget.html">Using CocoaWGet</a></li>
  <li><a href="html/wgethelp.html">The wget commands</a></li>
  <li><a href="html/wgetman.html">The wget man page</a></li>
</ul>
<hr>
```

The most important line in this file contains the `AppleTitle Meta` tag and its corresponding value `CocoaWGet Help`. The project uses this tag to find its online help files. The remaining section points to the other HTML files that complete the program documentation. Before continuing, look at the other files to get a sense of how they are constructed.

Creating an index file

The next step is to create an index file from your help files. The Help Viewer uses the index file to efficiently search your documentation. To create the index file, drag the CocoaWGet Help folder to the Apple Help Indexing Tool icon (/Developer/Application). The indexing tool processes your help files and creates a new index file called CocoaWGet Help idx in the help folder.

Adding help files to the project

Finally, add your help files to the project. To do so, set some key/value pairs that tell your program where to find the help files:

1. If necessary, open the CocoaWGet project.
2. Click on the Files tab and select the resource group from the Groups & Files pane.
3. Select Project→Add Files, navigate to the folder that holds the help files (CocoaWGet/CocoaWGet Help), and click the Open button.
4. Click the Create Folder References For Any Added Folders button and click the Add button.

This process adds the help files to the CocoaWGet project. Next, you need to provide the program with information that tells it where to find the help files and the names of the files:

1. Select the Targets tab, select the CocoaWGet target, and click on the Application Settings tab.
2. Under Basic Information, enter a unique name in the Identifier text field (org.book-example.CocoaWGet).

3 Click the Expert button and enter the following Propriety List/Value pairs, which define the location of the help folder and the help book name: `CFBundleHelpBookFolder` set to `CocoaWGet Help`; `CFBundleHelpBookName` set to `CocoaWGet Help`. The key `CFBundleHelpBookFolder` is the name of the folder that contains the bundle's help files, `CFBundleHelpBookName` is the name of the help file that Help Viewer presents when a user selects help.

4 Rebuild the program.

Finally, run the program and select the Help menu item. You will see the CocoaWGet help files displayed in the Help Viewer (see figure 6.17).

Figure 6.17
The main help window for the CocoaWGet program

As you can imagine, online help is usually more extensive than this example, including additional elements such as pictures, links, and even sounds. Because the help system is HTML-based, there are plenty of tools for creating help files. In addition, creating complex help is no harder than creating web pages.

6.8 Summary

This chapter has walked you through the steps of developing a fully functioning Cocoa program in Objective-C. You've learned how to design a program using the MVC design pattern, build a program's user interface in Interface Builder, and align interface elements using Interface Builder's built-in alignment feature. In addition, you have seen how to create classes, class instances, outlets, and actions, as well as how to connect these with interface elements.

You also learned to use Project Builder to write application code that handles the operation of your program. Another interesting element of this program is how it handles calling subtasks, in this case a UNIX command-line tool, to perform program actions. In addition, you saw how to reuse and modify existing code to address specific program features.

Cocoa, coupled with Project Builder and Interface Builder, provides a very convenient framework and environment for developing useful, expressive programs. After some experience, you will be able to develop programs quickly and efficiently to solve tasks in a variety of domains.

7 AppleScript programming

- Scripting languages
- Using the Script Editor and Script Runner
- Overview of the AppleScript language
- Developing an AppleScript for iTunes
- Developing an AppleScript Studio program

The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents.

—Nathaniel Borenstein

In chapter 6, you learned about programming under Cocoa, Apple's object-oriented framework for developing Mac OS X applications in Objective-C and Java. This chapter takes a different track, covering AppleScript, Apple's native scripting language used exclusively on the Macintosh. AppleScript offers Mac OS X users many advantages over traditional UNIX scripting languages and opens many new possibilities. The primary strength of AppleScript is in process automation. With AppleScript, you can automate many common tasks by using the services of one or more Mac OS X applications. This is a powerful idea, discussed in detail throughout the chapter.

This chapter covers the fundamentals of the AppleScript language, how to develop and run scripts, and some example AppleScript programs. Once you learn the basics, you will find AppleScript invaluable for automating many common Mac OS X tasks.

7.1 Introduction

Most UNIX developers are already familiar with scripting languages and probably know at least one that they use daily. Scripting languages, which are typically interpreted and dynamically typed, are extremely powerful for developing all sorts of programs from text file processing filters to software agents. As you'll recall, dynamically typed languages like Perl defer typing of data to the runtime system. On the other hand, statically typed languages such as C and C++ require you to provide type information at compile time, enabling the compiler to detect type problems before you run the program. In general, one method is not any better than the other, but rather more applicable to the problem you are trying to solve or the style of development you prefer.

For example, safety-critical software (such as medical applications) often requires formal reviews for program correctness, and therefore necessitates languages you can statically verify. Embedded systems often have hard performance constraints, calling for languages and tools that produce efficient compiled code. In these cases, statically typed languages like C and C++ are a logical choice.

7.2 Scripting languages

Scripting languages have existed since the 1960s. Early languages include JCL (Job Control Language), sh (the first shell), and Rexx; today's popular languages include Perl, Python, Ruby, and Tcl. Historically, UNIX systems have provided strong support for text processing, filtering, and automation through commands, pipes, shell scripts, and high-level scripting languages such as Perl or Python. The most basic technique for accomplishing these tasks is to use standard UNIX commands.

For example, to find how many processes a user is running you use the `ps` command to get a list of all running processes, `grep` to find all lines that contain the user name, and finally the `wc` command to count the number of lines:

```
% ps aux | grep omalley | wc –l
    25
```

Another technique is to use specialized tools, like `ed` and `awk`, which are designed for extracting and filtering lines of text. You generally use these programs together to perform tasks like filtering lines in a set of files and extracting and formatting information. Both UNIX commands and specialized tools provide you with primitives, but they do not give you the programmatic infrastructure to perform tasks that are more complex. Enter *scripting languages*.

Scripting languages like Perl and Python enable you to perform many of the tasks you accomplished using UNIX commands and tools, but they give you plenty of infrastructure to extend and enhance your solutions. In addition, these languages let you write programs that perform a wide range of tasks, from talking to remote hosts over a network or providing a GUI for user interaction, to performing mathematical operations.

There are many reasons to choose a scripting language over a compiled language for a project. Among them are the increased speed of development, the benefits of dynamic typing, and the high-level abstractions insulating you from low-level operations. The common cycle for developing programs in statically typed languages goes something like this: create and edit source files, build the code (compile and link), and run and debug the program. Over time, projects can grow to include many source files that may require recompilation for each new edit. Although there are techniques that reduce source dependency and thereby build times, the build phase can become very time consuming. Contrast this with development under scripting languages. Because these languages are interrupted, there is no build phase; development proceeds directly from edit to run. For mid-to-large projects, this process can greatly reduce the development cycle. In

general, compiled code does run faster than interrupted code, but for many applications, execution time is not a factor

Scripting languages also provide convenient abstractions for common operations. For example, Perl's `split` function places all elements of a line of text into an array, all in a single statement:

```
my @rec = split(/:/, $line);
print "$rec[0], $rec[1]\n";
```

Contrast this with a language like C. The C library provides primitives such as the `strtok` function to extract elements from a string, but you still need to implement the surrounding code to get the same functionality as `split`.

Scripting languages, as well as the UNIX commands and tools, are available under Mac OS X from the command shell. However, Mac OS X offers another scripting language that is specific to the Macintosh: AppleScript. AppleScript is a high-level scripting language that facilitates the manipulation of application and system services. The advantage of AppleScript over other scripting languages is that support for AppleScript is build into the Macintosh operating system, providing tighter integration with core system services, as well as utilizing the services and IPC facilities of Macintosh applications. In addition, AppleScript enables you to programmatically communicate with, and control, many Macintosh applications.

7.3 AppleScript

AppleScript supports many of the features of traditional scripting languages, but also includes features specifically designed for interacting with the Macintosh OS and Macintosh applications. Let's look at some general aspects of AppleScript and examine how it communicates with Mac OS X applications.

Imagine your Macintosh as containing an array of services encapsulated in various application programs. Programs such as BBEdit and Microsoft Word support text editing and manipulation services; programs like Fetch, Netscape, and Mozilla support network services for transferring files, logging in to remote hosts, and Web access; and others offer multimedia and audio facilities. Typically, users access application services by running the program and interacting with the GUI. For example, suppose you have a text document and wish to change all occurrences of the word *this* to *that*. To accomplish this task, you launch BBEdit, open the file, and choose the BBEdit service that replaces all occurrences of the word *this* with *that*. You can think of each of these actions (opening a file, searching and replacing text, and so on) as an application service that you access through the program's

GUI. Next, suppose you have replaced all occurrences and saved the file from within BBEdit. Now, you need to transfer the file to a remote host. Unless BBEdit supports this service, you are stuck. In this case, you need to use another program, such as Fetch (http://fetchsoftworks.com) or the secure copy command (`scp`, available from Terminal's command line), to send the file to the host.

However, what if the same application services you use from the program's GUI were available as a library, which you could access programmatically through AppleScript? Now, instead of being limited by the services of the program you are using, you would be free to write scripts that use and combine the services of many programs. Using the previous example, all you need to do is write a script that combines the text manipulation services of BBEdit and the network services of Fetch or `scp`. Now, in one script, you can automate and solve the required task. If you keep expanding this concept, you can see that the programs that come with your Macintosh, as well as any programs you add to the system (that support scripting), become participants in this game. You can accomplish many complex tasks that were otherwise impossible by using AppleScript to knit together application services.

In order for a script to use the services of a program, the author of the program must write it in a way that explicitly makes available application services for client access. These are called *AppleEvent-enabled programs*, meaning they can respond to requests from other programs by exposing services to the outside world. The underlying communication mechanism used to accomplish the interaction between clients and AppleScript-enabled programs is *AppleEvents*.

AppleEvents are defined messages that let applications extend their functionality by using the services of other applications and share their own operations with others applications. AppleScript communicates with applications by sending AppleEvents to other AppleEvent-enabled applications or system processes to request services and receive the results of the operation.

For example, AppleScripts communicate with an AppleEvent-enabled program by sending an AppleEvent to a program and receiving an AppleEvent as the result of the operation. The developer of a Macintosh program chooses what services to export to clients and implements these services. Typically, these services are the same that are available from within the application, but they can be a sub- or superset of those services. Apple suggests that Macintosh applications support at least four Standard Suite events: open, print, quit, and run.

Let's look at an example of these events in action. Here's a simple script that uses some of the basic AppleEvents:

```
tell application "BBEdit 6.5"
  activate
  open {file "Macintosh HD:Users:omalley:shuttle:junk.txt"}
end tell
```

First, the script launches the BBEdit program using `activate`; if the program is already running, it makes the program active. Next, the script opens the specified file. (Do not worry about how to run this example; you will learn more about it in the next section.)

7.3.1 Creating and running a script

Let's get a feel for AppleScript by developing some basic scripts. The AppleScript folder, located in Application/AppleScript, contains two main programs that you will use to develop scripts: Script Editor and Script Runner.

Script Editor

Script Editor is the main script development environment for writing AppleScripts. You use the program to write and run an AppleScript during development (see figure 7.1).

You use the description text area for entering and storing information about the AppleScript. To hide this window, click on the disclosure triangle next to the description label. The lower part of the window contains a text area for entering the AppleScript. Above this window are four buttons: Record, Stop, Run, and Check Syntax:

- *Record*—Lets you record the AppleScript commands associated with a sequence of actions for a particular program. For example, to record commands, click the Record button, switch to an application that supports recording, and perform your actions. Once you are finished, switch back to the

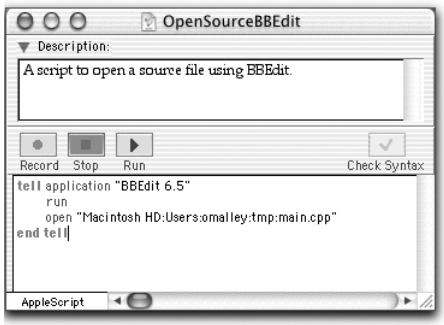

Figure 7.1
The Script Editor is the main development environment for writing an AppleScript.

Script Editor and click the Stop button. You will see the AppleScript code for the executed commands displayed in the script's text area. Remember, not all applications are recordable; if the Script Editor does not display code, the program is probably not recordable.

- *Stop*—Stops the execution of a script.
- *Run*—Executes the AppleScript commands contained in the script text area.
- *Check Syntax*—Enabled by the Script Editor when you enter new code and check it for syntax errors.

Script Runner

The Script Runner program acts as an aggregator of scripts, giving you a launching pad for your AppleScripts (see figure 7.2). To run Script Runner, open /Application/AppleScript and double-click on the Script Runner icon.

Clicking on the program's main window displays the application menu (also shown in figure 7.2). This menu lists the folders on your system that contain AppleScripts, as well as their contents. AppleScripts are stored in /Library/Scripts and your home directory, within ~/Library/Scripts.

Keeping scripts in folders is a good way to organize and access related scripts from the Script Runner menu. In addition, I prefer to order the menu starting with my scripts (whose folder is prefixed with an underscore) followed by system-level scripts. To add a new script to Script Runner, exit the program, open a script folder (/Library/Scripts or ~/Library/Scripts), drag the compiled script to the target folder, and launch Script Runner.

In general, Script Runner is a nice way to organize completed scripts that you access often. (Try adding it to the Dock for better access.)

7.3.2 Types of AppleScripts

There are two main types of AppleScripts: *compiled scripts* and *applets*. A compiled script requires the Script Editor or Script Runner to run, cannot be viewed from within a text editor, and therefore must be edited from Script Editor. Applets are self-contained, stand-alone programs that do not need the Script Editor to execute. To run these, you simply locate the program and double-click on its program icon. In addition to the main types of AppleScripts, you can also create applications called *droplets*. Droplets, as the name suggests, are scripts that execute when you drag an object to their icon. Droplets are great for scripts that process files or the contents of a directory. When you drop a file on a droplet, it processes the file; when you drop a folder, it iterates over all items in the folder.

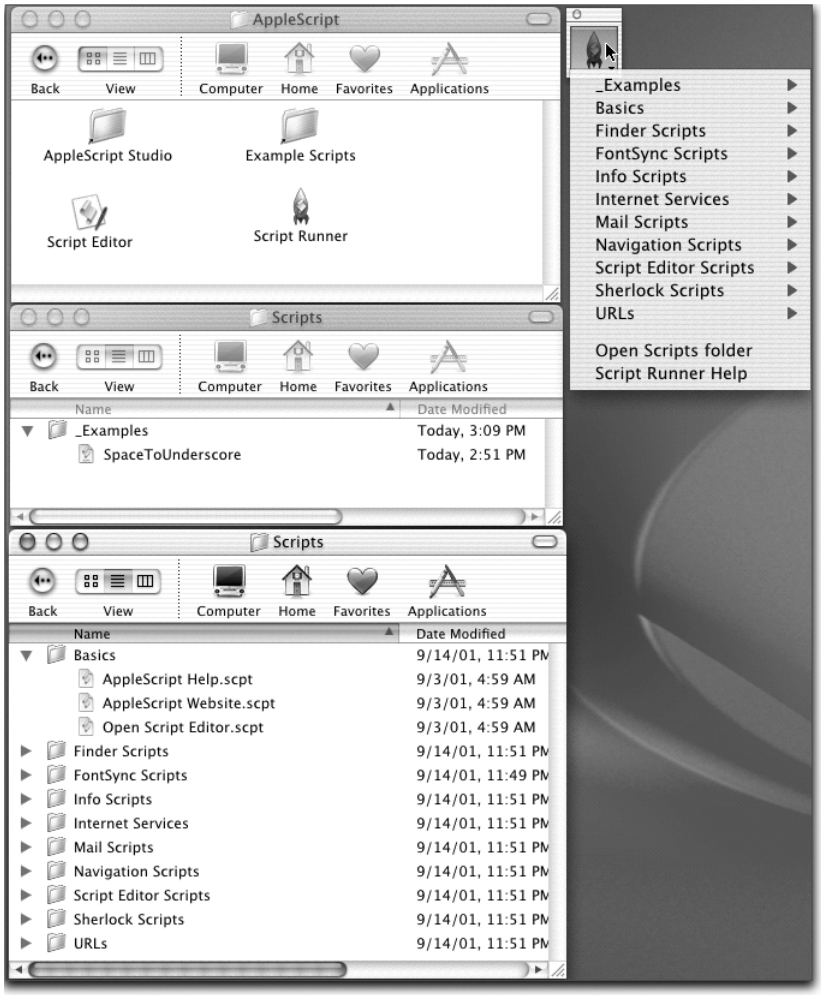

Figure 7.2 Script Runner is a convenient way to manage your AppleScripts, enabling you to collect and launch your scripts from a single location.

7.3.3 *AppleScript extensions*

Most scripting languages have user communities that are constantly developing new scripts and tools. For example, if you are a Perl programmer, you are likely aware of the Comprehensive Perl Archive Network (CPAN). CPAN is a huge collection of freely available Perl software, modules, and documentation that you can download and use in your programs. These modules cover a broad spectrum of

domains including database interfaces, mail and usenet news, and development support. If you are thinking of developing a Perl program for a particular domain, some part of it probably already exists in CPAN.

AppleScript supports language extensions through *scripting additions* (also called *osaxen* for their code type, Open Scripting Architecture eXtension). Developers write scripting additions in C or C++; once they're compiled and installed, you can use them in your AppleScripts.[1] Scripting additions have many advantages, including faster execution times and access to system-level resources. For example, AppleScript has a function called `display dialog` that displays a dialog box that can hold information from an AppleScript. The `display dialog` call is a scripting addition. Basically, if AppleScript does not have the functionality you need for your program, you can add it with a scripting addition.

Under Mac OS X, scripting additions are stored in the /System/Library/ScriptingAdditions folder (see figure 7.3). To install a scripting addition, drag it to this

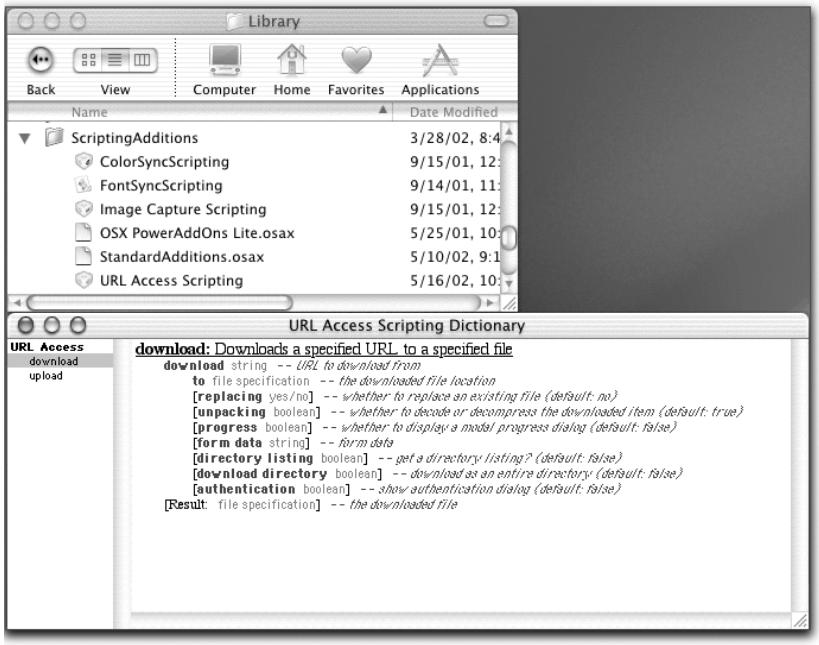

Figure 7.3 Scripting additions extend the AppleScript language and are stored in /System/Library/ScriptingAdditions.

[1] For more information about writing your own scripting additions, see Apple Technical Note TN1164 (http://developer.apple.com/technotes/tn/tn1164.html#DoRunX).

folder (you need root privileges to add items to this folder). To see what services are available from a scripting addition, open the Script Editor and select File→Open Dictionary. Select the scripting addition from the list and click the Open button.

7.3.4 The AppleScript language

Compared to other languages, AppleScript looks more like English than code. In general, AppleScript is a dynamically typed, English-like scripting language that supports many of the features of standard UNIX scripting languages but is geared to the Macintosh and is primarily used for application automation and control. As such, it's stronger in these areas than as a general-purpose programming language.

AppleScripts consist of a series of statements. You structure these statements as *commands* that operate on *objects*. A command is an AppleScript statement that requests an action of an object. Objects are either application objects or system objects. Application objects are associated with a particular application, covering the services to which the application will respond. System objects are associated with the Mac OS X system. You manipulate objects with structured or procedural code. In AppleScript, you can view application programs as object hierarchies.

For example, an application's object hierarchy is laid out in its dictionary, which is viewable from the Script Editor. Figure 7.4 shows the dictionary for TextEdit. The left window lists the application's commands and objects; commands are in plain text, and objects are italicized.

Commands operate on objects. For example, the command `open` operates on objects such as a window or document. Conceptually, this is similar to object-oriented

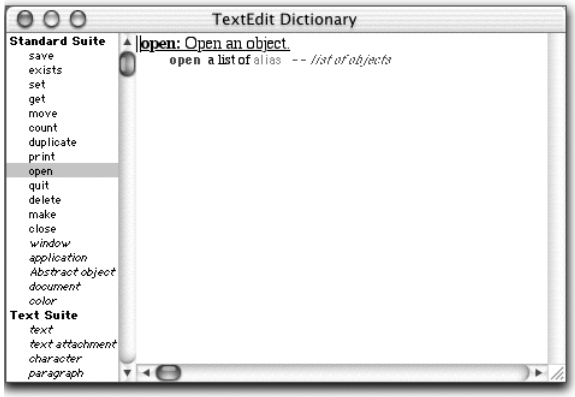

Figure 7.4
The dictionary for TextEdit lists the application's commands and objects; commands appear in plain text and objects in italic text.

programming, where you send a message to an object that performs the appropriate action.

In AppleScript, you scope commands and objects using the `tell` statement:

```
tell application "TextEdit"
  quit
end tell
```

Data types and data structures

AppleScript supports the usual data types, such as the integer, real (float), string, and boolean, as well as structured types such as Date, List, and Record. Because the language is dynamically (loosely) typed, you do not need to specify the type of a variable at development time. Table 7.1 shows the syntax and an example of the most commonly used data types.

Table 7.1 AppleScript data types and examples

Type	Description	Example
Integer	Unsigned, nonfractional number	`set n to 10` `get n`
Real	Unsigned, fractional number	`set x to 1.123` `get x`
Number	Either an Integer or a Real	(see Integer and Real examples)
Boolean	Logical value, true or false	`set aBool to false`
String	Series of 1-byte characters	`set msgStr to "This is a message string"`
Text	Same as String	(see String example)
Date	String that holds a date	`set now to date "5/22/02"`

List	Collection of values, which can be of any data type (array)	`Set aList to {x, y, z, 1, "as"}`
		`-- get the first items in the list`[2]
		`item 1 of aList`

Record	Collection of properties (hash)	`Set js-bach to {name: "JS-Bach", period:"Baroque", birth:1685, death:1750}`
		`-- get Bach's birth date`
		`birth of js-bach`

[2] In AppleScript, -- is used to indicate a comment.

If you like, you can explicitly specify a type of a variable as follows:

```
set n to 100 as integer
get now as string
```

The List data type is an array, which enables you to hold data of different types. Unlike in C, Lists begin at index one, not zero. In addition, you can nest Lists within Lists:

```
set aList to {n, x, aBool, msgStr, date}
set aNewList to {aList, {1, 2, 3, 4}}
```

The following script demonstrates how to perform some simple math operations in AppleScript:

```
set total to 0

set aList to {1, 2, 3, 4, 5, 6, 7, 8, 12, 45, 67, 54, 12,
              4, 12, 12, 45, 56}
repeat with n in aList
  set total to total + n
end repeat

set mean to total div (length of aList)
-- log is not the mathematical log, but rather a log,
-- or print statement.
log "length of list: " & (length of aList)
log "sum of list: " & total
log "mean of list: " & mean
```

In addition to these data types, AppleScript defines many constants such as `pi`, `true`, `false`, `space`, `return`, and `tab`. See the AppleScript documents for a complete list of these constants.

In figure 7.5, you can see the output of the previous script. (`log` is useful for performing `printf`-style debugging of a script. To see the output of a `log` command, open Controls→Open Event Log and make sure you enable Show Events and Show Event Results.)

Control and iteration statements

Control statements affect the execution path through a script. AppleScript supports the usual assortment of control and iteration statements found in other languages, including `if`, `else if`, `repeat`, and `exit`.

if and else if

The `if` and `else if` control statements are used to define which statements are executed based on some condition. Table 7.2 lists some examples.

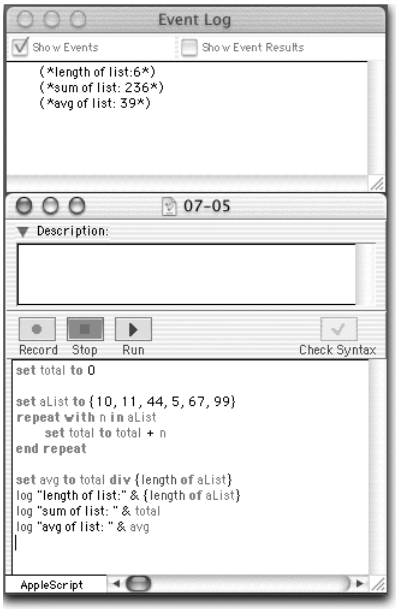

Figure 7.5
An example of the `log` command, which is used to perform `printf`–style debugging of a script

Table 7.2 Examples of `if`, `else if` statements

`-- Single if statement`
`if x > max then`
`  max = x`
`end if`
	`-- Single if/else if state-`
	`  ment`
	`if x == 1 then`
	`  y = 1`
	`else if x == 2 then`
	`  y = 2`
	`else if x == 3 then`
	`  y = 3`
	`else`
	`  y = -1`
	`end if`
		`-- if/else statement with`
		`  logical operators`
		`if x > y and x < z then`
		`  y = x`
		`else if x > y or y < z then`
		`  y = z`
		`end if`

Loop/iteration statements

Loops, or iteration statements, enable programs to repeatedly call a sequence of statements. Table 7.3 contains some examples.

Table 7.3 Examples of AppleScript iteration statements

```
-- Infinite repeat
set i to 0
repeat
  set i to i + 1
    if i _ 100 then
      exit repeat
    end if
end
```

```
-- repeat k times
set i to 0
repeat 100 times
   set i to i + 1
end
```

```
-- repeat until
set done to false
set i to 0
repeat until done
  set i to i + 1
    if i _ 100 then
      set done to true
    end if
end repeat
```

```
-- repeat while
set done to false
set i to 0
repeat while not done
  set i to i + 1
    if i _ 100 then
      set done to true
    end if
end repeat
```

```
-- repeat from 1 to k
set total to 0
repeat with n from 1 to 100
  set total to total + n
end repeat
```

```
-- Iterate over a list
set aList to {1, 2, 3, 4, 5,
   6, 7, 8}
repeat with n in aList
  log n
end repeat
```

In addition to the standard control and loop statements, AppleScript supports some other constructs specific to the language. These include `tell`, `try`, and `timeout`. Table 7.4 lists these statements and examples.

Table 7.4 Miscellaneous AppleScript statements

```
-- tell
tell application "Text-
   Edit"
  quit
end tell
```

```
-- try
try
  -- statements.
On error errmsg
-- handle error
end try
```

```
-- with timeout
with timeout of 10 seconds
  -- statements
end timeout
```

Operators

Table 7.5 lists the AppleScript arithmetic, logical, and comparison operators.

Table 7.5 AppleScript operators

Type	Operators
Arithmetic	+, -, *, /, ^, mod
Logical	and, or, not

Table 7.5 AppleScript operators *(continued)*

Type	Operators
Comparison	=[3] equal, equals, equal to, is, is equal to != does not equal, is not, is not equal to < comes before, less than, is less than > comes after, greater than, is greater than ≤ is less than or equal to, less than or equal ≥ is greater than or equal to, greater than or equal

Subroutine/function calls

Like most programming languages, AppleScript supports function (subroutine) calls. Here is an example of how to use a subroutine call in AppleScript:

```
set rc to getMax(1, 4)
log "1, 4, max val is: " & rc

set rc to getMax(14, 4)
log "14, 4, max val is: " & rc

-- Subroutine that returns the maximum value of the pair.
on getMax(x, y)
  if x > y then
    return x
  else
    return y
  end if
end getMax
```

In addition to calling subroutines located within the current script, you can also call subroutines located in other scripts:

```
tell application "theApp"
  activate
  [subroutine]
end tell
```

[3] String comparisons are not case sensitive.

Comments

AppleScript uses two comment syntaxes: single- and multiline comments. Single-line comments are preceded by two hyphens (`--`). Multiline comments begin with the `(*` token and end with a `*)`:

```
-- This is a single line comment
(* This is a multi-line comment that extends over¬
more than on line *)
```

Another feature in this example is the use of the ¬ character for continuing a line. To insert this character, press Option-Return.

Input/output

AppleScript excels at program automation. However, sometimes you need to use it as a general-purpose programming language. One of these cases is text-file processing. Like it or not, a simple, structured text file is one of the best ways to store data and log data during automation tasks. Here are some basic file I/O operations:

```
-- Open a file.
try
  set fp to open for access file (myFile) with write permission
on error
  log "error opening: " & myFile
end try

-- Close a file.
try
  close access fp
on error
  log "error closing file"
end try

-- Read from a file.
try
  set buf to (read fp)
on error
  log "error reading file"
end try

-- Write to a file.
try
  write s to fp
on error
  log "error writing file"
end try
```

XML-RPC and SOAP

One of the coolest (and simplest) things to do with AppleScript is to access network-based services using XML-RPC and SOAP. XML-RPC (remote procedure call) permits machines to communicate over a network and share and exchange services using XML for message encoding and HTML as a transport. SOAP (Simple Object Access Protocol) also exchanges information over a network, but it uses an object-based hierarchy and requires named parameters (see http://www.w3.org/TR/SOAP for more information).

You should get used to the idea of writing and using services over a network. Thinking this way expands the possibilities of your programs by tapping into a wealth of existing network-based services. In addition, it enables you to build distributed applications using remote services and write services that others can access. See the XMethods site (http://www.xmethods.net/site) for a list of available services as well as more information on using SOAP.

Script debugging

As you develop more scripts, you will sometimes need to trace the execution of your script, examine the contents of a variable, or perform some action to determine why a script is failing—known as *debugging*. AppleScript and the Script Editor support some limited facilities for debugging your scripts. Before we look at them, let's briefly discuss some common debugging techniques that apply to AppleScript.

If you have ever developed asynchronous, network-based applications, you know how difficult it can be to understand the cause of a failure. For this reason, a common debugging technique is to use trace files to synchronously log the sequence of program statements. By keeping your messages structured, you can easily parse out the files and see what's going on in a program. The following listing shows an example of this technique in the context of an AppleScript, along with the contents of the output log file:

```
property DEBUG_LOG : ((path to startup disk as text) & "debug1.log")

-- Clear the log file
set fp to open for access file (DEBUG_LOG) with write permission
set eof fp to 0
close access fp

tell application "BBEdit 6.5"
  my TRACE_INFO("activate")
  activate
  my TRACE_INFO("make new text window")
  make new text window
  my TRACE_INFO("set bounds " & "{0, 0, 200, 300}")
  set bounds of text window 1 to {0, 0, 200, 300}
```

```
end tell

TRACE_WARNING("This is a warning message.")
TRACE_ERROR("This is an error message.")

on TRACE_INFO(msg)
  set fp to open for access file (DEBUG_LOG) with write permission
  set now to current date
  write "I:" & now & ":" & msg & return to fp starting at eof
  close access fp
end TRACE_INFO

on TRACE_WARNING(msg)
  set fp to open for access file (DEBUG_LOG) with write permission
  set now to current date
  write "W:" & now & ":" & msg & return to fp starting at eof
  close access fp
end TRACE_WARNING

on TRACE_ERROR(msg)
  set fp to open for access file (DEBUG_LOG) with write permission
  set now to current date
  write "E:" & now & ":" & msg & return to fp starting at eof
  close access fp
end TRACE_ERROR

-- Example output from the above code.
I:Sunday, June 9, 2002 11:39:29 AM:activate
I:Sunday, June 9, 2002 11:39:29 AM:make new text window
I:Sunday, June 9, 2002 11:39:29 AM:set bounds {0, 0, 200, 300}
W:Sunday, June 9, 2002 11:39:30 AM:This is a warning message.
E:Sunday, June 9, 2002 11:39:30 AM:This is an error message.
```

Embedding these trace statements in your code enables you to easily trace the execution of your scripts.

Another interesting, through less practical, technique uses the `say` command. The `say` command instructs AppleScript to speak the enclosed statement.

For example, the following code shows a use of the `say` command:

```
say  "opening file " & myFile
try
    set fp to open for access file (myFile) with write permission
  on error
   say  "error opening file " & myFile
end try
```

Yet another method is to use the `display dialog` command to display debugging information in a dialog box:

```
tell application "BBEdit 6.5"
  display dialog "activate"
  activate
```

```
  display dialog "make new text window"
  make new text window
  display dialog "set bounds " & " {0, 0, 200, 300}"
  set bounds of text window 1 to {0, 0, 200, 300}
end tell
```

Each `display dialog` call will display a dialog box with the quoted text.

One of the best and most practical debugging techniques is to use the AppleScript's `log` command in conjunction with the Script Editor's Event Log window. The `log` statement prints a log message to the Event Log window, which is accessible from within the Script Editor (Controls→Open Event Log [Command-E]). Checking the Show Events checkbox will log all AppleEvents to the window. Checking Show Event Results displays log messages.

The following script produces the logging events shown in figure 7.6:

```
tell application "BBEdit 6.5"
  log "calling activate"
  activate
  log "calling make new text window"
  make new text window
  log "setting bounds to " & "{0, 0, 200, 300}"
  set bounds of text window 1 to {0, 0, 200, 300}
end tell
```

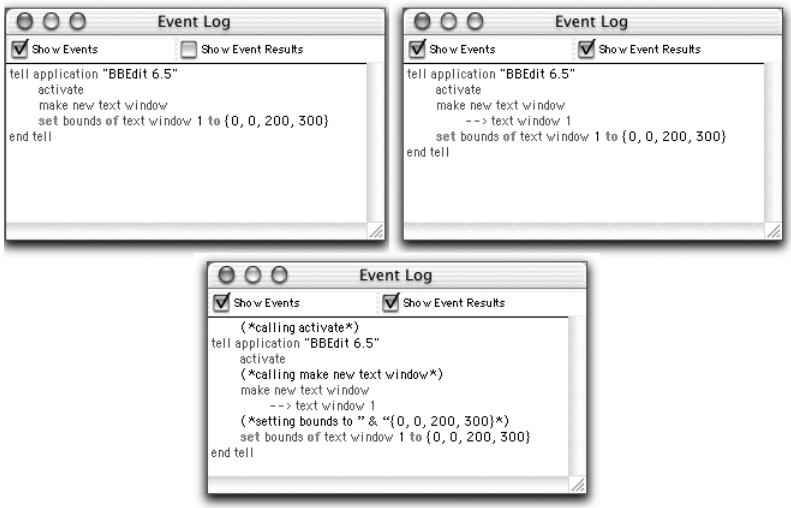

Figure 7.6 The Event Log displays the output of a script and the AppleScript `log` command. The first window shows the result of the script (minus `log` statements) with the Show Events checkbox selected. The second window shows the output when Checking Show Event Results is checked (minus `log` statements). The third window shows the results of the embedded `log` commands.

7.3.5 Choosing a scripting language

As you have seen from the previous discussion, AppleScript is a powerful, loosely typed scripting language. It supports many of the constructs you would expect from a scripting language, including a rich set of data types and control statements. Even though you can use it for many common programming tasks, its primary use is automating, controlling, and tying together Macintosh applications. For general programming problems such as text-file processing, database access, and network programming, Perl, Python, and Ruby are better choices. However, if you need to do any scripting that requires the services of Macintosh programs, AppleScript is the right language.

In addition to writing pure AppleScripts, you can achieve more power and functionality by combining it with other languages using AppleScript Studio. AppleScript Studio enables you to add Cocoa-based GUIs to your scripts and to combine scripts with Objective-C, providing more power and options.

In addition, you can combine AppleScripts with other scripting languages by sending AppleEvents to the Terminal application and having it run your UNIX-based scripts. This technique is demonstrated later in the chapter when I discuss AppleScript Studio and show how to use AppleScript to invoke `gnuplot`.

7.4 Example applications of AppleScript

In this section, you will learn to write two types of AppleScript-based programs. The first program demonstrates using AppleScript to control the services of iTunes, Apple's digital music program.(With iTunes, you can easily play recorded music from CDs or MP3s, organize you music collection, convert your CDs to MP3 format, and burn CDs of your favorite MP3s.) The second script is an AppleScript Studio program that you can use to monitor a program's memory usage and plot the results.

7.4.1 iTunes and AppleScript

If you are anything like me, you own lots of CDs, MP3s, and even LPs. With the advent of iTunes and cheap, large hard drives, I have taken to converting many of my CDs to MP3 format and storing then on my hard disk. Doing so makes it easy to transport music from my desktop machine to my laptop or iPod, or to burn mixes of MP3s to CD. The problem is, as my music collection expands, it's difficult to keep track of what I have and where it is located. The iTunes program solves some of this situation through *playlists*—lists of related songs that help you organize related songs in your music collection under a name. However, you need to create each playlist by hand. (The current [as of this writing] version of iTunes has a feature called Smart Playlists, which automates the creation of custom playlists.)

Using AppleScript, you can easily automate many iTunes tasks. Apple's iTunes web site (http://www.apple.com/itunes) contains a downloadable collection of useful scripts for iTunes 2.0 and greater. The collection includes scripts to build a CD tray insert from a play list, export a summary of the library, or play random tracks. As useful as these are scripts are, they are more useful to study and use as the basis for your own scripts. The script collection contains a script called Make Playlist By Artist, which creates a single playlist containing all the tracks for a particular artist. You will use it as the basis for your first script and extend it so it iterates over the entire library and creates playlists for each artist or album. (The iTunes library contains a list of all tracks you've imported from CDs or downloaded from the Internet, or that exist on your hard drive. Basically, the library contains your entire music collection, which can be viewed in different ways.) Before getting into this script, let's look at some of the AppleEvents supported by iTunes.

iTunes scripting services

To see what scripting services iTunes supports, Open the Script Editor, select File→Open Dictionary, and choose iTunes from the list (see figure 7.7).
The left window lists the scripting services iTunes supports. All scriptable applications export a dictionary of services that developers can read from the Script Editor. As I pointed out earlier, commands are in plain font and objects are italicized. To get more information about a command or object, click on its name. For example, figure 7.7 shows the result of selecting the `duplicate` command, which duplicates one or more objects of a given type.

When you begin using AppleScript, an application's dictionary may seem a bit abstract and terse. At first, you should use the dictionary along with sample code

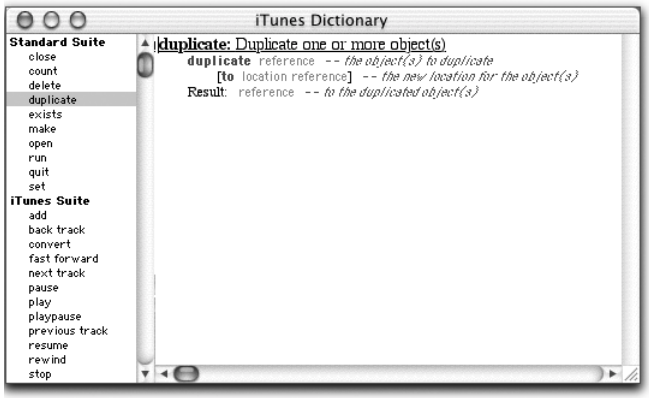

Figure 7.7
The iTunes dictionary displays the commands and objects that are accessible to AppleScript.

when scripting a Mac OS X application. As you gain more experience, the dictionary will make more sense, enabling you to use it directly as you would an API.

You will notice iTunes' menu bar, which contains an AppleScript menu. Many Mac OS X applications offer this service, which lets users run scripts directly from within an application. To install scripts into iTunes, open the Library folder in your home directory (~/Library), create a folder called Scripts, and add your AppleScripts to this folder. When you relaunch iTunes, the AppleScript menu will appear and display the scripts in the Scripts folder. As I mentioned earlier, Apple provides many useful scripts that you can download and use.

Writing a script

Now that you understand the basics, let's move on to creating your own script that you can use within iTunes. You will use parts of the Make Playlist By Artist script as the basis of a new script that creates playlists for all artists or albums in your iTunes library. Before continuing, copy the new script (located on the chapter07 folder of the source distribution), called Make Playlist, to the iTunes script folder (~/Library/iTunes), and relaunch iTunes.

Here's how this version works:

1. Select the script from the iTunes scripts menu. It asks if you wish to delete all current playlists. Doing so is useful if you wish to create new, unique playlists from your library.
2. The script asks if you want to create playlists by artists or album. Choose either option.
3. The script loads all records in the library into a list, sorts them, and iterates over the list, creating playlists based on the user selection. For larger collections, this process can take a few minutes.

Now, let's look at some of the highlights of the script to give you a feel for how it works, as well as how it interacts with iTunes. The first block of code comes directly from the original script. Its purpose is to check the current version of iTunes:

```
set this_version to the version as string
    if this_version is not greater than
       or equal to the required_version then
      beep
      display dialog "This script requires iTunes version: " &
                     required_version & ¬
        return & return & ¬
        "Current version of iTunes: " & this_version buttons
        {"Update", "Cancel"} default button 2 with icon 2
      if the button returned of the result is "Update" then
```

```
        my access_website("http://www.apple.com/itunes/download/")
        return "incorrect version"
      end if
    end if
```

You can use and modify this code in other scripts to make sure that users are running the correct version of AppleScript. The only change I made was to move the code to a subroutine and pass in the expected program version.

The next section of code prompts the user, asking if it should delete all playlists. This code is a good example of a reoccurring theme in AppleScript programming—prompting the user for information and using control statements to perform the correct action:

```
display dialog "Delete all playlists?" buttons
{"Yes", "No", "Cancel"} default button 1
    if the button returned of the result is "Yes" then
      Display dialog "Deleting all playlists…"
      buttons {"•"} default button 3
                    giving up after 1
      my delete_playlists()
    end if
```

The first statement displays a dialog box containing three buttons (Yes, No, Cancel) and sets the default button (highlighted button) to Yes. Next, the script checks the result of the user selection using the `if` statement. If the user clicked the Yes button, the script displays another dialog box saying it is deleting all playlists. Notice the last part of this statement: `giving up after 3`. You use this statement to control how long the script displays a dialog box if no user interaction is detected. In this case, if the user does not click the button, the script displays the dialog box for three seconds. Finally, the script calls the `delete_playlists` subroutine to delete all playlists.

Another common operation prompts a user to enter information, such as a test string:

```
display dialog "Enter some text:" default answer ""
set the theText to the text returned of the result
```

Another interesting block of code is the `sort` subroutine, which comes directly from the Apple code. This code takes a list, sorts it, and returns the list to the caller. Take a look at the syntax of the call:

```
set the the_list to my ASCII_Sort(the_list)
```

This statement says to set `the_list` to the value returned by `ASCII_Sort`.

Next, let's look at the `create_playlists` subroutine, which is used to create the playlists. This code is an example of some common operations: iterating

over a list and interacting with iTunes. I've removed unnecessary code from the listing to make it more readable (see the actual script for the full version):

```
on create_playlists(the_list, type)
  tell application "iTunes"
    repeat with i from 1 to (number of items in the_list)
      set this_item to (item i of the the_list) as string

      set this_playlist to make new playlist
      if type is equal to "Artist" then
        set the name of this_playlist to this_item
      else if type is equal to "Album" then
        set the name of this_playlist to "_" & this_item
      end if
      tell source "Library"
        tell playlist "Library"
          if type is equal to "Artist" then
            duplicate (every track whose artist is this_item)
            to this_playlist
          else if type is equal to "Album" then
            duplicate (every track whose album is this_item)
            to this_playlist
          end if
        end tell
      end tell
    end repeat
  end tell
end create_playlists
```

The first line, `tell application "iTunes"`, scopes the calls that follow to the iTunes application. Next, the script iterates over the list (`the_list`), using the counter `i` as an index into the list data structure, storing the result in the variable `this_item`. It sets the name of the new playlist to this value. It also prefixes each album playlist with a `"_"` to help distinguish and group them from artist playlists.

The highlighted code shows the statements that call iTunes services. The first statement creates a new playlist using the `make` command from the iTunes Standard Suite. The other statement also uses the iTunes Standard Suite's `duplicate` call to copy the names of the matching individual tracks to the playlist.

In general, this script performs the required actions and provides a good example of the power of AppleScript. However, it could be improved. For example, for large sets of playlists, the delete operation can take some time; creating playlists is also time consuming. Another real limitation is the fact that the script cannot update existing playlists with new tracks; instead, it deletes all playlists and creates new ones from scratch. Most of these issues are easily addressed and are left as an exercise (how's that for passing the buck?).

7.4.2 AppleScript Studio

If you use traditional scripting languages such as Perl and Python, you know of their usefulness in developing support applications. However, their support for developing modern user interfaces is weak. Because these programs are used primarily by experienced users, GUIs are often not necessary. Both languages offer modules and libraries for user interface work, but the user interface support is very limited. Basically, if you are using these languages, you run your scripts from the command line.

If you would like your scripts to reach a wider audience of users without experience in programming and command-line tools, you should give them a modern user interface. This is exactly what you get with AppleScript Studio—it lets you write the guts of your application in AppleScript and construct your interface using the Cocoa frameworks. You can produce AppleScript-based applications that look and behave exactly like any other Mac OS X application.

The name AppleScript Studio can be a bit of a misnomer, because it is not a separate development tool. To construct AppleScript-based applications, you use the already-familiar Project Builder and Interface Builder. Here is a high-level view of how it works:

1. Launch Project Builder, select File→Create New Project, and select one of the AppleScript Studio project templates.
2. Open the Resource folder using the discloser triangle and double-click on the project's MainMenu.nib file to open the file in Interface Builder.
3. Construct your application's user interface and name each interface component you wish to access from your scripts (more on this later).
4. Create new script-based handlers for interface actions. The development environment creates these handlers and adds them to the main AppleScript of the application.
5. Move back to Project Builder and fill in each handler with AppleScript code that responds to interface actions.

The process is obviously more detailed than this, but these are the main steps for creating AppleScript Studio programs. If you have ever used other script-based development environments such as HyperCard, SuperCard, or MetaCard, these steps will sound familiar. Basically, you draw the interface, attach handlers to interface components, and implement each handler in a scripting language.

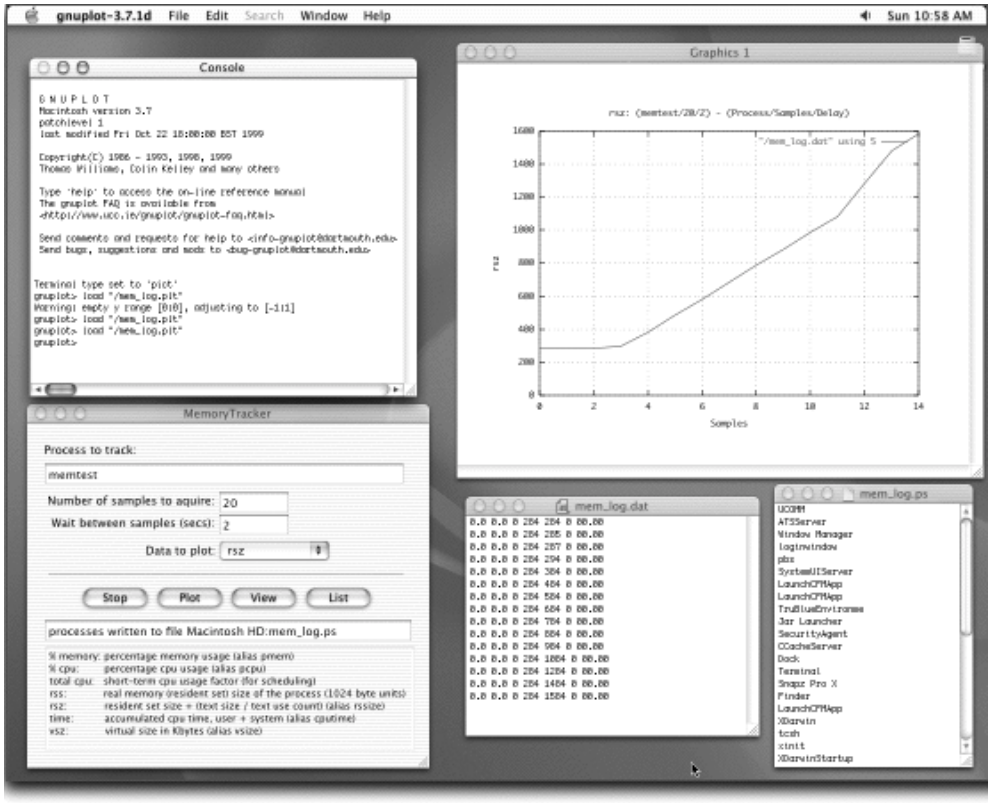

Figure 7.8 All application screens for the MemoryTracker program

Let's look at an AppleScript Studio application called MemoryTracker (see figure 7.8). MemoryTracker is implemented entirely in AppleScript and makes extensive use of other applications' services through AppleEvents. MemoryTracker's purpose is to monitor a running application's memory usage and display a plot of various memory-related items.

To use the program, you enter a running program's name in the Process To Track text field. Click the List button to open a window displaying all running processes (lower-right application window in figure 7.8). You use the Number Of Samples To Acquire field and the Wait Between Samples field to set how many samples the program acquires and its sample rate. Once these parameters are set, click the Start button to begin the process of acquiring data. After the program acquires data, you select the information you wish to plot from the Data To Plot

menu and click the Plot button. Doing so plots the selected channel in `gnuplot`.[4] Finally, clicking the View button displays a window of the raw acquired data.

As you can see from the description, this simple program has many features, most of which already exist as either command-line tools or Mac OS X applications. For example, to obtain information about a running program, you can use the command-line tool `ps`. To parse and format the output of `ps`, use `awk`. To display data and process lists, use the text editor TextEdit; and to plot data, you use the Mac OS X version of the venerable UNIX tool `gnuplot`. To tie these components together, you use AppleScript and the Cocoa frameworks.

Other than `gnuplot`, all these components come standard with the Mac OS X system. If you did not use existing tools, you would have to write these components by hand, which is time consuming. Because they already exist, you can easily build this program in a few hours.

Opening the project

Project Builder makes creating AppleScript-based projects a snap by providing three AppleScript Studio project templates: AppleScript Application, AppleScript Document-Based Application, and AppleScript Droplet. Our application uses the AppleScript Application template.

The MemoryTracker project is located in the source_code/chapter07/MemoryTracker folder. Locate this directory from the Finder and double-click on MemoryTracker.pbproj to launch Project Builder and load the project.

WARNING If, like many UNIX users, you have the command-line version of `gnuplot` installed, but not the Mac OS X version described in this section, you may get the cryptic error message "Application.applescript:140: No user interaction allowed. (-1713)" when you build the project.

Building the interface

The first step in creating the application is to construct the user interface in Interface Builder. The general cycle is to create your application's interface component, such as a window, populate it with interface controls by dragging each control from the Cocoa Views palette to the window, and connect the appropriate controls to program elements such as outlets and actions.

---

[4] Available from http://homepage.mac.com/gnuplot.

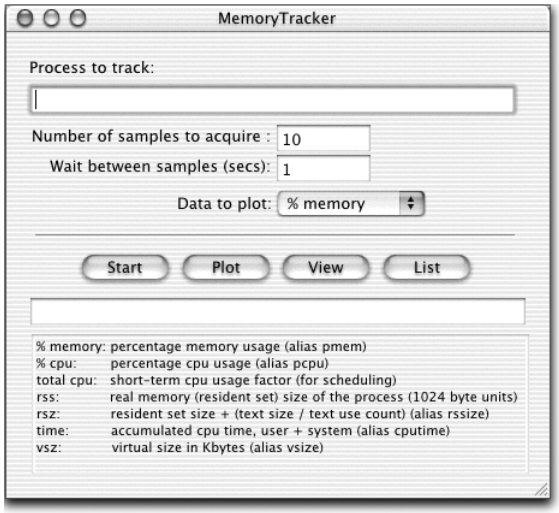

Figure 7.9
The main window for the MemoryTracker program

Let's look at the user interface for the program. Open the Resource group in the Groups & Files pane and double-click on MainMenu.nib to launch Interface Builder and open the file. Next, click on the Instance tab and double-click on the Window icon.

I built the MemoryTracker user interface by dragging each interface component from the Interface Builder palette to the appropriate location within its tabbed view (see figure 7.9).

Table 7.6 lists the control types for the various interface objects.

Table 7.6 The data types for the various interface objects of the MemoryTracker program

Item	Type
Process to track (as well as the other window labels)	System Font Text
All test fields	`NSTextField`
Buttons	`NSButton`
Data To Plot menu	`NSPopupButton`
Help reference	`NSTextField` with Small System Font Text

Next, look at the main script handlers and see how interface components are associated with handler code. Click the Start button, choose Tools→Show Info, and select AppleScript in the popup menu in the palette (or press Command-6) to display AppleScript from the popup menu (see figure 7.10). Note that the Action and Clicked options are selected, as well as the Application.applescript box.

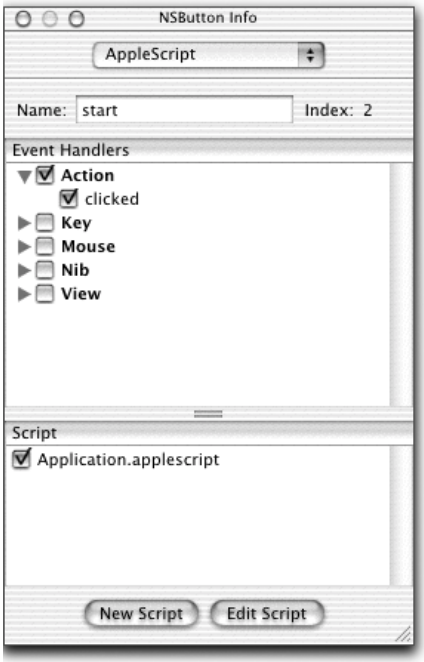

Figure 7.10
Use the Attributes dialog to associate a script handler with an interface element.

Also, note that the Name text field is set to `start`. This name is associated with the handler script `clickedStart`, which responds to a user clicking the Start button. To view the script, click the Edit Script button to display the script handler in Project Builder. You can repeat these steps to see the attributes and code associated with the other interface items.

Table 7.7 lists the names for each interface component.

Table 7.7 The names of the interface components

Item	Name
Process To Track	process
Number Of Samples To Acquire	samples
Wait Between Samples (Secs):	sleepsecs
Data To Plot menu (NSPopupButton)	datatoplot
Start, Plot, View, List	`start`, `plot`, `view`, `list`
Status field	status

Now that you understand how Interface Builder connects interface objects to handlers, let's look at the AppleScript code. Switch back to Project Builder and open the MemoryTracker.applescript file.

Let's begin by looking at the main handler, called `clicked`:

```
on clicked theObject
  tell window "MemoryTracker"
    set processName to contents of text field "process"
    set nSamples to contents of text field "samples"
    set nSleep to contents of text field "sleepsecs"
    set dataToPlotPos to contents of popup button "datatoplot"
    set dataToPlotName to title of current menu item
        of popup button "datatoplot"
  end tell

  if theObject is button "start" of window "MemoryTracker" then
    my clickedStart(theObject, processName, nSamples, nSleep)
  else if theObject is button "view" of window "MemoryTracker" then
    my clickedView()
  else if theObject is button "plot" of window "MemoryTracker" then
    clickedPlot(dataToPlotPos, dataToPlotName,
    processName, nSamples, nSleep)
    activate
  else if theObject is button "list" of window "MemoryTracker" then
    clickedList()
  end if
end clicked
```

This handler is the entry point into all handlers for the program. It is passed a single parameter called `theObject`. You use this object to determine what action the user took and to call the appropriate code to handle the request. The first block, enclosed in `tell window "MemoryTracker"`, gets the values from the interface and places them into the appropriate data members. Note that the name in double quotes refers to the name you gave to each interface component. Next, the script uses the object to determine which button the user clicked.

As you can see, the syntax for this script is very English-like. For each button, it calls the proper AppleScript subroutine to handle the details of the request. Notice the handling code for the Plot button: after plotting the channel, it calls the `activate` statement so the MemoryTracker application comes to the front of the screen. Because the user typically does not interact with the plot (except perhaps to save or print the plot), this action saves the user a click click when choosing another channel.

The `getMemory` subroutine gets information on a process. This function shows how easy it is to talk to the Terminal application and run command-line tools with AppleScript. In this case, it uses the command-line tool `ps` to get specific

information on all running processes, `grep` to find the process it is interested in, and `awk` to format the input:

```
on getMemory(cmdToken)
  set psCmd to "ps axo %mem,%cpu,cpu,rss,rsz,time,vsz,ucomm
               | grep " & cmdToken & " | grep -v grep
               | awk '{print $1 \" \" $2 \" \" $3
               \" \" $4 \" \" $5 \" \" $6 \" \" $7\" \" $8}'"
  set theResult to do shell script psCmd
  STATUS_MSG(psCmd)

  if theResult is equal to "" then
    STATUS_MSG("returned empty string")
    set memList to {}
    return memList
  end if

  set mem to 1st word of theResult
  set pCpu to 2nd word of theResult
  set cpu to 3rd word of theResult
  set rss to 4th word of theResult
  set RSZ to 5th word of theResult
  set t to 6th word of theResult
  set VSZ to 7th word of theResult

  set memList to {mem, pCpu, cpu, rss, RSZ, t, VSZ}
  return memList
end getMemory
```

The first two lines of the function show how easy it is to chain these tools together, execute the command, and return the result to a variable. These simple tools make it easy to debug, as well. For example, if you enter a process ID, rather than a program name, in the entry box, the program will not find anything. It is easy to type **`ps axo %mem,%cpu,cpu,rss,rsz,time,vsz,ucomm`** in a Terminal window to see just what is being returned, and to then determine that the name, not the pid, is the proper value for the field.

The format of the return value is a string of space-separated values. Next, the function sets each returned value to its corresponding variable, adds them to a list, and returns the list to the caller.

The script calls this function repeatedly from the `runGrabMemory` function. This function first clears the file used to store the process data and iterates for the number of samples requested by the user. On each iteration, it gets the process data by calling `getMemory` and writes the data to a file. Once complete, it closes the file and returns:

```
on runGrabMemory(processName, nSamples, nSleep)
  set fp to open for access file (DATA_FILE_MAC)
  with write permission
```

```
  set eof fp to 0
  close access fp

  repeat with cnt from 1 to nSamples
    set msgPrefix to processName & ":" & nSamples & ":" & nSleep
    set memList to getMemory(processName)
    set fp to open for access file (DATA_FILE_MAC)
    with write permission

    set el to "processing sample: " & cnt & "/" & nSamples
    STATUS_MSG(el)
    repeat with n in memList
      set el to el & "."
      STATUS_MSG(el)
      write n & " " to fp starting at eof
    end repeat
    write return to fp starting at eof
    close access fp
    set el to "sleeping: " & nSleep & " (secs)"
    my sleep(nSleep)
  end repeat
  STATUS_MSG("done processing")
end runGrabMemory
```

Another interesting function is `plot`, which displays a plot of the user-selected channel:

```
on plot(pos, theName, processName, nSamples, nSleep)
  set pos to pos + 1
  set msg to "calling gnuplot to plot " & theName
  set plotCmd to "plot " & "\"" & DATA_FILE_UNIX
  & "\" using " & pos & " with lines"
  set s to "plotting data for " & theName
  & ", process " & processName

  -- Clear the file.
  set fp to open for access file (PLOT_FORMAT_FILE_MAC)
 with write permission
  set eof fp to 0
  close access fp

  set fp to open for access file (PLOT_FORMAT_FILE_MAC)
  with write permission
  write "set xlabel " & "\"Samples\" " & return to fp
  set s to "\"" & theName & "\""
  write "set ylabel  " & s & return to fp


  set s to theName & ": (" & processName
  & "/" & nSamples & "/" & nSleep & ") - (Process/Samples/Delay)"
  write "set title  " & "\"" & s & "\"" & return to fp
  write "set yrange [ 0 : ]" & return to fp
  write "set grid" & return to fp
```

```
    write "set key right" & return to fp
    write "set data style linespoint" & return to fp
    write plotCmd & return to fp
    close access fp

    tell application "gnuplot-3.7.1d"
      activate
      -- exec "load \"/mem_log.plt\""
      set s to "load \"" & PLOT_FORMAT_FILE_UNIX & "\""
      exec s
    end tell
  end plot
```

This function is a good example of how to use the functionality of a scriptable Mac OS X program to accomplish a task. If you have ever written a graphing program, you will appreciate being able to use the services of another program rather than writing your own. For plotting, you use a Mac OS X version of `gnuplot`, available from http://homepage.mac.com/gnuplot. As mentioned earlier, building the project will give an unhelpful error message if the scriptable Mac OS X version of `gnuplot` is not installed.

The function opens a text file, writes a plot file based on the user selection and plot data, and sends `activate` and `exec` AppleEvents to `gnuplot`. Doing so causes `gnuplot` to become the active application: it loads the plot file and displays the plot.

The View and List buttons use the services of the text editor TextEdit, which comes standard with Mac OS X. Clicking the View button causes the scripts to activate TextEdit and open the file that contains the raw process data. Clicking the List button calls `getAllProcesses`, which runs the `ps` command to get the names of all running processes. Next, the script activates TextEdit and opens the file that contains process name information:

```
on clickedView()
  tell application "TextEdit"
    activate
    open {(DATA_FILE_MAC) as alias}
  end tell
end clickedView

on clickedList()
  getAllProcesses()
  tell application "TextEdit"
    activate
    open {(PROCESSES_FILE_MAC) as alias}
  end tell
end clickedList
```

Look at the rest of the code for the program. You will find it straightforward and easy to understand.

As this example program illustrates, you can implement powerful tools with AppleScript in almost no time. Your AppleScript programs can be complex, requiring GUI-based user interfaces, or simple in-house tools that you can use to support application development. In either case, AppleScript Studio makes these things possible and is an enjoyable environment to work in.

This example also demonstrates the advantages of using AppleScript over traditional UNIX scripting languages. Yes, you can produce the same workflow with Perl, but you cannot package the program under a single icon or develop a Mac OS X Aqua-based user interface. In addition, UNIX scripting languages do not enable you to use the services of other Mac OS X programs.

7.5 Summary

This chapter introduced you to the important concepts of AppleScript and walked you through the steps of developing two AppleScript programs. The first example program used Script Editor (the main script development environment for writing AppleScripts) and iTunes (Apple's digital music player) to show how to interact with the services of a Mac OS X application.

The other program used AppleScript Studio. AppleScript Studio enables you to write AppleScript-based applications with a Cocoa GUI. These programs look and behave exactly like any other Mac OS X applications but are predominantly written in AppleScript.

8 Mac OS X and beyond

- New features of Jaguar
- Additions to the developer tools
- Project Builder features
- Terminal application features
- PerlObjCBridge

I never think about the future. It comes soon enough.

—Albert Einstein

Now to look at the more interesting features UNIX developers can expect from Jaguar. This chapter provides information about the new developer tools and infrastructure, covers the new Terminal application, discusses the new features of Project Builder, and briefly shows you some of the features of the PerlObjCBridge.

8.1 Introduction

On March 24, 2001, Apple released the first version of Mac OS X (v10.0) to the public. Later that year, it released version v10.1, which focused primarily on performance improvements. Since then, most of the updates to the operating system have centered on bug fixes and feature enhancements. In late summer 2002, Apple released its first major upgrade to Mac OS X, code-named Jaguar. Jaguar builds on the previous versions of the system but adds many new features and enhancements; according to Apple, it includes more than 150 new features (http://www.apple.com/macosx).

A number of Jaguar's new features focus on changes and updates to the core operating system, many of which will be of interest to UNIX users. For example, Jaguar now uses the Common UNIX Printing System (CUPS) for printing. CUPS uses the Internet Printing Protocol (IPP) to maintain print jobs and provides better access to modern printer features than older printing systems like the Berkeley Line Printer Daemon (LPD).

Jaguar includes compatibility with FreeBSD 4.4, Kerberos authentication, and `gcc` version 3.1, and more support for porting UNIX programs to the Macintosh by improving compatibility with the POSIX API, libraries, and headers. There is also a new and improved Terminal application that supports more options and emulation modes. Another nice addition for Perl programmers is PerlObjCBridge, developed by Doug Wiebe of Apple, which enables Perl code to access Cocoa objects, register as clients for notifications from Cocoa-based frameworks, and do messaging between Perl scripts running on different machines.

In addition to the changes under the hood, Jaguar extends many of the features of current Mac OS X applications. The Mail, Address Book, and Sherlock applications have been revamped with additional features and services. The new Finder improves performance by threading tasks and contains a find feature so you can search for files without leaving the current window. Several new

applications and technologies also appear, including an AOL-compatible instant messaging client, called iChat QuickTime 6, Quartz Extreme, Rendezvous, Inkwell, and Sherlock 3.

One of the most interesting new features is Rendezvous. The goal of Rendezvous is to simplify the process of setting up networks and sharing services between machines. Basically, it enables hosts on an IP network to discover one another and share services. For users, this means no more network setup or configuration—they can simply plug in a new device on the network (a computer or printer, for example) and let it configure itself.

Another new feature is Quartz Extreme. Quartz is the primary display technology for Mac OS X. Quartz Extreme extends Quartz by offloading graphics screen rendering operations from the CPU to the graphics card. These means faster screen update and operations, and less use of the CPU. To use this technology, you must have a supported graphics card.

In the user space, many of the new features are designed to tie together the services of various programs, enabling seamless sharing of data between applications. In addition, Address Book stores data in vCard format and can access information from LDAP servers.

8.2 Development tools

The developer tools for Jaguar come with many new features. This section lists some of the most important and useful additions.

8.2.1 Compilers

The developer tools for the pre-Jaguar version of Mac OS X use `gcc` 2.95 as the default compiler. Jaguar ships with `gcc` 3.1 (http://gcc.gnu.org/gcc-3.1). This new version of `gcc` contains many improvements and fixes to the earlier version of the compiler: better language standards conformance, a faster and more memory efficient preprocessor, and improvements in the C++ library implementation.[1] In addition, this release includes better support for profile-directed optimization. Let's examine this feature.

Optimization

Many optimization techniques attempt to improve a program by either making its code smaller or making its performance faster. Choosing the right option often

[1] See http://gcc.gnu.org/gcc-3.1/changes.html and http://gcc.gnu.org/onlinedocs/libstdc++/faq/index.html#4_1.

means picking the appropriate tradeoff between these two goals. Many optimizers statically scan intermediate code, looking for obvious inefficiencies based on some rule set. For example, the optimization technique called *dead code elimination* detects and removes unreachable code (code that will not affect the results generated by a program). In the following example, the code in the `if` block is never executed and can be safely removed:

```
int
main()
{
  bool x = false;
  if (x) {
    printf("some text");
  }
  // Rest of program...

  return 0;
}
```

Another common optimization is *common subexpressions elimination*. Here, the compiler detects and eliminates expressions that are unnecessarily calculated:

```
// Contains an unnecessary calculation.
int x = 10.09
int y = 1.99;
float v[1000];
for (int i=0; i<1000; i++) {
  v[i] = x * y + i;
}
// Optimized.
int z = 10.09 * 1.99;
for (int i=0; i<1000; i++) {
  v[i] = z + i;
}
```

When optimizing code, the compiler will implement a heuristic to determine what optimization to perform.

Profile-driven optimization

The compiler can make better optimization choices if it understands the runtime behavior of the program it's optimizing. Using this information, the optimizer can decide on the most efficient optimization strategy for the program. This technique is called *profile driven optimization*, and is a feature of `gcc` 3.1 (this feature was contributed by Jan Hubicka of SuSE Labs, together with Richard Henderson of Red Hat, and Andreas Jaeger of SuSE Labs).

As the gcc 3.1 documentation points out, most profile-driven optimization techniques use a two-pass implementation. The first pass generates information about the runtime behavior of the program, which the optimizer uses on the second pass to inform its optimization choices. This process requires more diligence on behalf of the user because of the extra step involved in generating runtime behavior data. Additionally, running the program such that it generates useful profile data can be tricky (http://gcc.gnu.org/news/profiledriven.html).

To address these issues, gcc implements profile-driven optimization differently. Instead of using dynamic data from past runs, it uses a static branch predictor that infers program execution behavior. In many situations, static profiling techniques provide a reasonable tradeoff between the traditional two-pass, time-consuming technique and the faster, less, time-consuming method.

8.2.2 Project Builder

The current (as of this writing) version of Project Builder (2.0.1) contains some new and useful additions. These include using gcc 3.1 as its default compiler, inline scripting in the style of MPW-worksheets, a new target editor, better debugging support, and batch searching of developer documentation. Let's look at some of these additions and features.

8.2.3 Changing compilers

As I discussed in the first part of this section, the new development tools include gcc 3.1, which Apple recommends you use instead of the gcc 2.95 compiler—Project Builder will use gcc 3.1 as its default compiler. If for some reason you wish to use the gcc 2.95 compiler, click on your project's Targets tab, select a target, click on GCC Compiler Settings, and select GCC v2.95.2 from the Compiler version menu (see figure 8.1).

8.2.4 Inline scripting

As you learned in chapter 3, the Macintosh Programmer's Workbench (MPW) was one of the first development environments for the Macintosh. In many ways, MPW was a UNIX-like development environment that combined a command-line environment with elements of an IDE. To perform operations, you entered commands into worksheets, or used the GUI interface. This combination (command-line and GUI interface) made MPW a very powerful and extendable development environment.

Project Builder now includes inline scripting in the style of MPW. The two main additions are as follows:

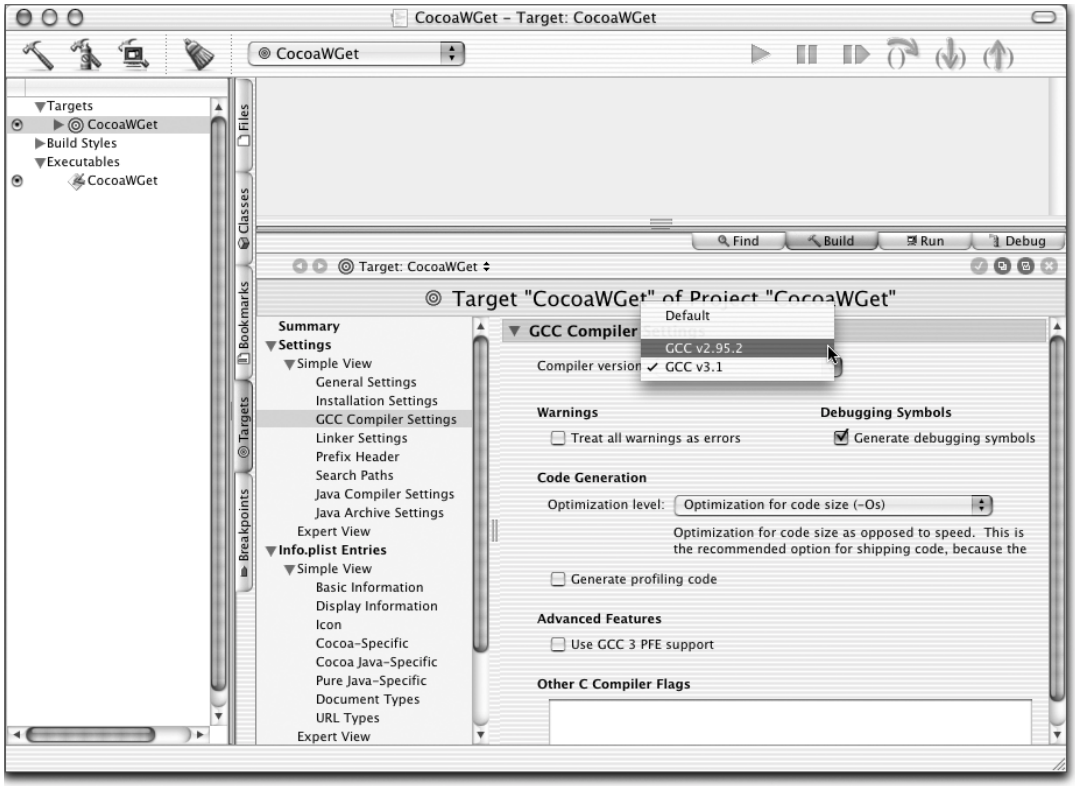

Figure 8.1 Project Builder now uses `gcc` 3.1 as its default compiler. You can switch back to `gcc` 2.95 using the target preference's GCC Compiler Settings.

- Running shell commands inline, or within a Project Builder text buffer
- Running shell scripts from a customizable menu from within Project Builder

Let's look at each of these features in more detail.

Running shell commands from within a Project Builder text buffer is a very powerful feature and opens up all sorts of interesting possibilities. This feature encompasses a full range of shell commands such as `ls`, `pw`, and `cal`, and running Perl statements from the command line (`perl -e`) or running Perl scripts. The inline-scripting feature does not support running commands that would require user interaction, such as `vi` or `emacs`.

To execute a shell command, place the cursor in a text buffer, type the command(s), and press Control-R (the cursor can be anywhere on the line where the command appears). The output of the command is written after the command.

For example, imagine you are testing a server program and wish to check if it is accepting connections correctly. One way to check this is to write a test agent that sends a fixed number of messages to the server. The server contains code to log all connection information to a file. After running several tests, you wish to see if the server accepted all client connections—if the agent sent 100 messages, the server should have accepted all 100. A simple check is to count the number of connections in the server log file. Figure 8.2 shows how to do this from within Project Builder using inline scripting.

You can save a lot of time by running commands from within Project Builder rather than jumping to a shell. In addition, the results are saved to the buffer for later use.

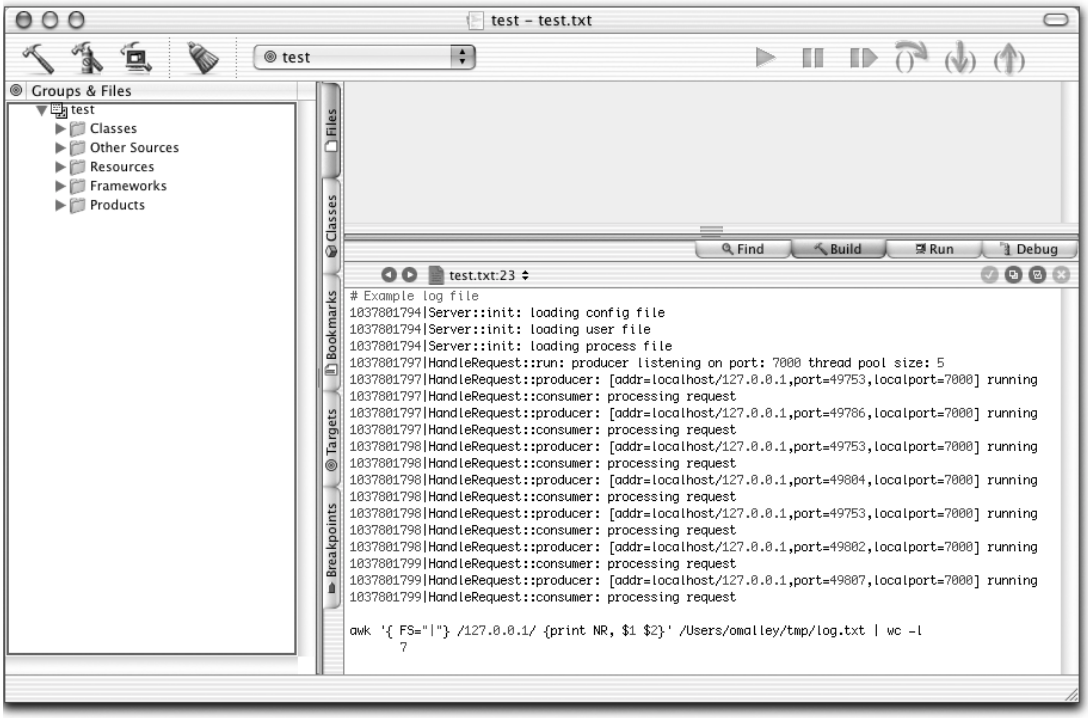

Figure 8.2 The new inline scripting feature enables you to run shell commands and scripts from within a Project Builder text buffer.

NOTE Project Builder lets you set many of its defaults through its user interface (using Project Builder→Preferences). However, many more are settable using the `defaults write` command from the Terminal application:

```
defaults write com.apple.ProjectBuilder <defaultName>
<defaultValue>
```

For example, to save the dynamically generated jamfiles to disk during builds, set `PBXSaveJamfiles` to `YES`.

```
% defaults write com.apple.ProjectBuilder PBXSaveJamfiles YES
```

Project Builder must be restarted for the changes to become active. For more information, select Help→Show Expert Preferences Notes from within Project Builder.

In addition to running scripts in a text buffer, you can also store scripts in files and run them from a menu within Project Builder. To add a script menu to Project Builder, copy all files from /Developer/ProjectBuilder\ Extras/ExampleScripts/ to ~/Library/Application\ Support/Project\ Builder and restart Project Builder if necessary.

Project Builder's Application Support folder (~/Library/Application\ Support/Project\ Builder) now contains two additional items: a file called StartupScript and a folder called Scripts. The StartupScript file is a shell script that adds the script menu to Project Builder. Because it is a shell script, you can add other shell commands to it if you like. The Scripts folder contains a set of shell scripts that are added to the User Scripts menu by StartupScript. You can add scripts to this folder, and they will be added to the script menu when Project Builder is launched (see figure 8.3).

As you can imagine, being able to run and display the result of a shell command within Project Builder is a powerful feature. Remember, each time you run a shell script or command, Project Builder executes a new shell; so, there are some limitations on what you can do. For more information about scripting, see Help→Show Release Notes.

8.2.5 New target editor

Project Builder's GUI has been updated to support a new target editor. In the older version of Project Builder (pre 2.0.1), the Target view holds two display panes: Targets and Build Styles (see figure 3.12). The Targets pane displays all targets for the project; clicking on a target shows its settings. Build styles are displayed in

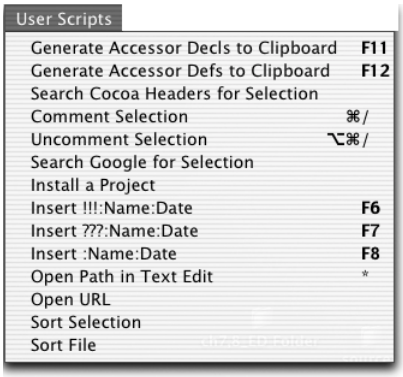

Figure 8.3
Any scripts you save to the Project Builder Application Support folder (~/Library/Application\ Support/Project\ Builder) are added to the User Scripts menu when Project Builder is launched.

a subpane, where you can view and edit their settings. In addition, the Executable view (seen by selecting the Executable tab) displays the different executable programs your application contains.

The new version of Project Builder combines targets, build styles, and executables into a single view and presents the information in a table format. In addition, the Executable tab has been removed, and its features have been combined with the Target view. Figures 8.4 and 8.5 show the new layout of the target editor and the executable features.

8.2.6 Searching documentation

Project Builder supported batch searches of project code, Frameworks, and open files through the find tab. Using this feature, you could perform textural and regular expression searches of your files. The current version of Project Builder (as of this writing) supports a new search type—the content-based documentation search. This search type permits you to perform content searches of the Mac OS X Developer Documentation, a handy feature that will save you lots of time.

Searches can be in the form of natural language questions such as "Show me information on Darwin" (figure 8.6) or common strings like "AppleScript".

In addition to these items, Project Builder contains many other enhancements. Debugging features have been improved. Within a debugging session, you can now Control-click on a valuable in the Debug Variables view to display a contextual menu with several new addition. The View Value As… addition lets you cast the currently selected variable to a new type. Doing so will change the variable and open a new window that displays the variable and its value. Once the variable goes out of scope, Project Builder closes the window.

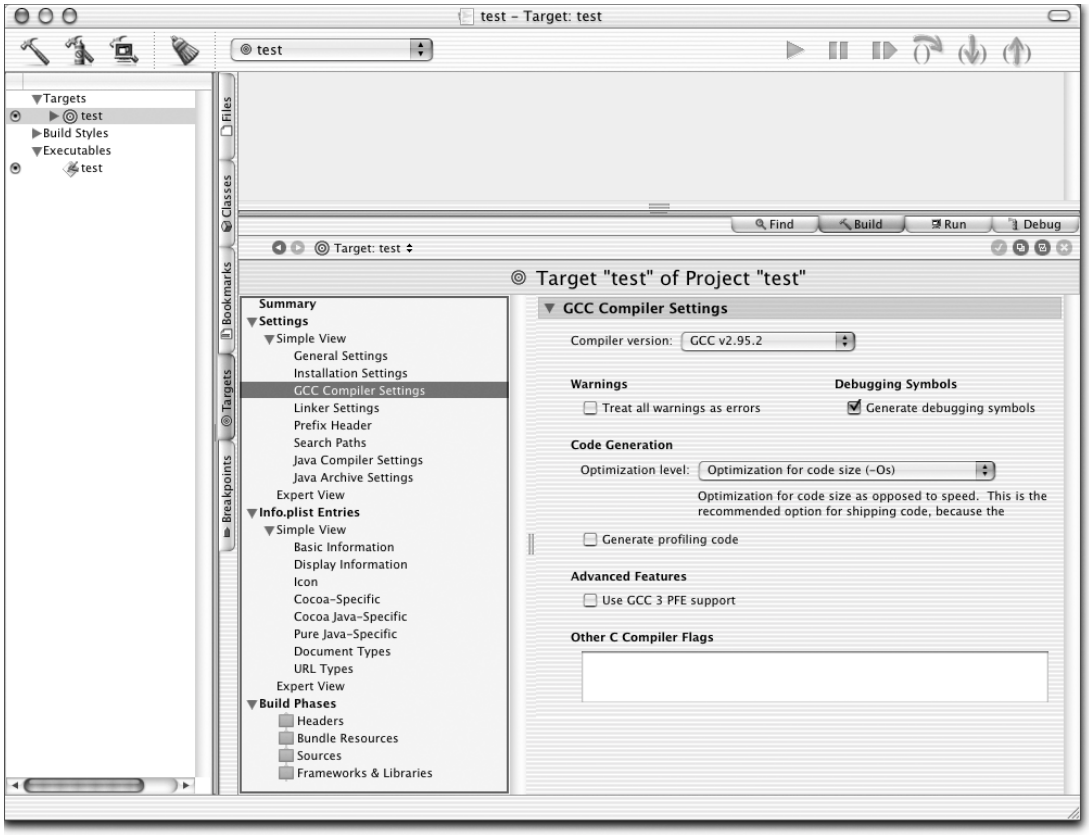

Figure 8.4 The new target editor displays options in a table, enabling you to access and alter settings.

You can now split editor panes side-to-side as well as top-to-bottom, use new keyboard commands to pop the various navigator bar menus (Ctrl-L displays the Loaded Files pop-up, Ctrl-2 displays the Function pop-up, and Ctrl-3 displays the Included Headers pop-up) and copy the results of a find operation using Command-C or by highlighting and dragging the text to targets that accept textual pastes. See Help→ Show Release Notes for more information on the Project Builder's new features.

Along with the changes to the interface, several new features are included, such as a new Summary module that displays a target's name and type, and a comment field for adding more information about the target. In addition, you can now construct a build style to install builds from within the IDE. As always, see the release notes for more information.

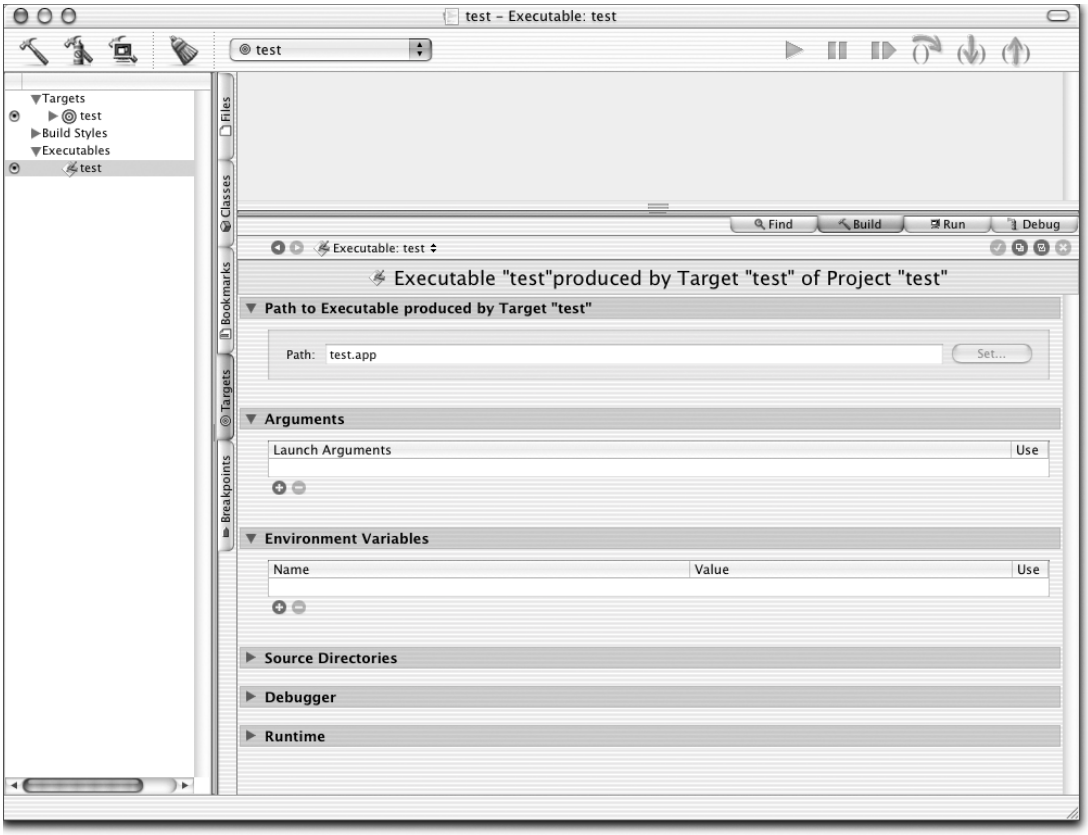

Figure 8.5 In the new version of Project Builder the Executable tab has been removed; its features are merged into the Targets tab settings.

8.3 Terminal application

If you use Mac OS X primarily as a UNIX box, you probably spend a lot of your time with the Terminal application. Terminal implements a command interpreter, or shell, which enables you to interact with the system through its UNIX interface. The Terminal application that ships with Jaguar adds some new features you may find useful.

8.3.1 Setting Terminal preferences

The first big change is the way you set preferences. In previous versions, all preferences were accessible from a single Preferences dialog box (see figure 8.7).

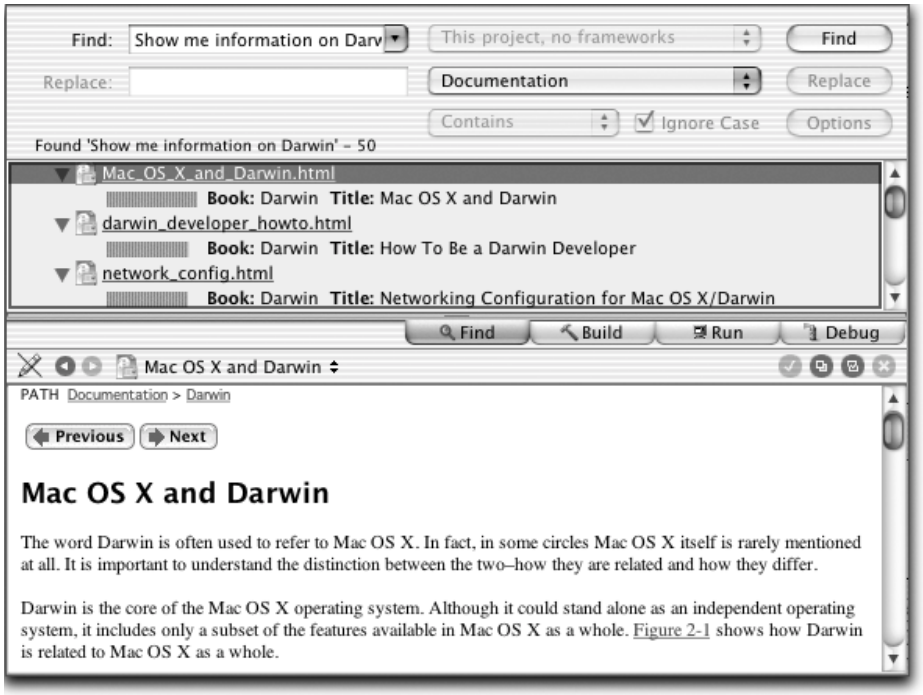

Figure 8.6 Project Builder now let you search developer documentation using the content-based documentation search feature.

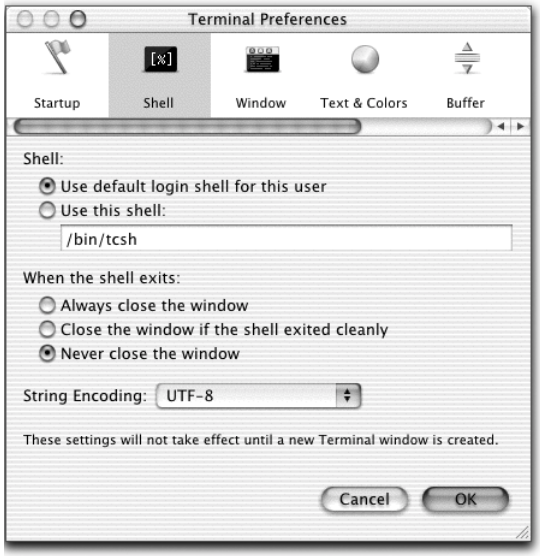

Figure 8.7
The Terminal application that comes with v10.1.x aggregates preferences into a single dialog, enabling you to get at them in one place.

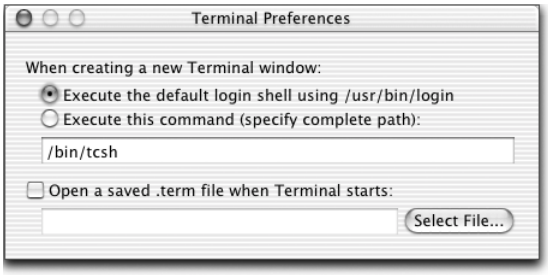

Figure 8.8
The new Terminal Preferences dialog lets you add actions to the shell on startup.

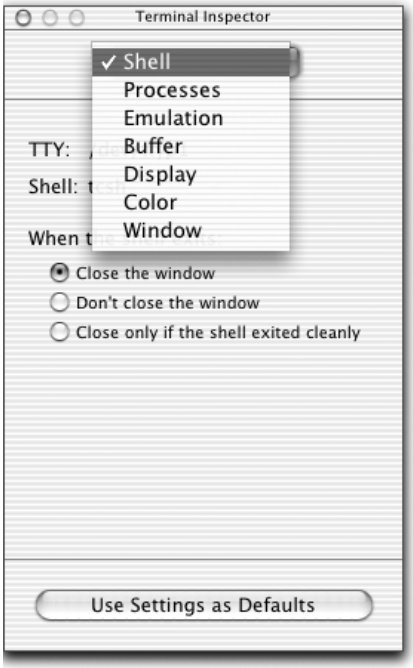

Figure 8.9
Many of the preferences from the older Terminal are now available from the Terminal Inspector.

The new Terminal application splits preferences between two menu items: Terminal→Preferences (figure 8.8) and Terminal→Window Settings (figure 8.9).

You use the Terminal Preferences dialog box to add behavior to the shell on startup. The Terminal→Window Settings menu item opens a floating dialog box (Terminal Inspector) that you use to set window options, including your shell, buffer size, window colors, and window properties. One improvement of this version is that preferences immediately apply to the active window; in past versions, you had to open a new shell to see the changes. To save your selected preferences, select File→Use Settings As Defaults.

8.3.2 Splitting the Terminal window

Another new addition is being able to split a Terminal window. When you click on the split icon, located in the upper-right part of the window, the Terminal window is split in half. The upper pane holds a scrollable, read-only history of your Terminal session. The lower pane contains an active area where you can continue working. You can use this feature to save the history of a program and compare it to the current run.

For example, suppose you wish to visually trace the memory usage of a program you suspect is leaking memory. In one shell, you run the leaky program. In another, you run `top`.

After a few iterations, click on the split icon in the window running the `top` command and drag the horizontal bar to the middle of the window. As you can see in figure 8.10, you can view the past instance of `top` in the upper pane at the

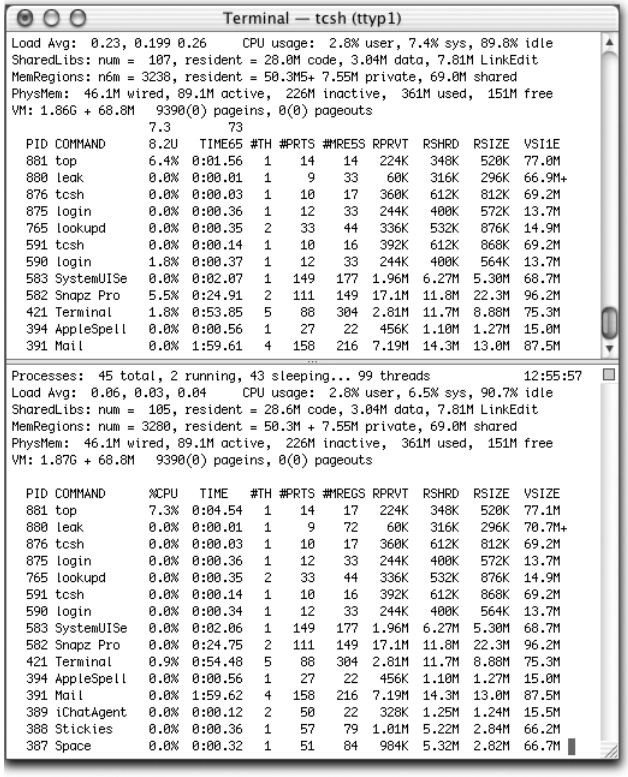

Figure 8.10
Being able to split the Terminal window lets you view past Terminal histories while continuing your work.

same time `top` is running. By splitting the window, you save screen space and the bother of running another shell.[2]

8.3.3 Other Terminal additions

In addition to these features, here are some other additions:

- You can copy text between Terminal windows by highlighting the text in one window and dragging it to the other. You can also drag text from a Terminal window to another application, or from an application to the Terminal window.
- You can make the Terminal window transparent by choosing Terminal→ Window Settings and selecting Color from the pop-up menu. This option is not very useful for editing, but in some cases (say, when screen real estate is limited) you may want to layer windows in such a way that you can see information through each shell.
- The new Terminal supports antialiasing of fonts and better vt100/vt220 emulation.

8.4 The PerlObjCBridge

As the last few chapters demonstrate, Project Builder and Interface Builder are good development environments for creating Mac OS X applications. These tools, coupled with the Cocoa frameworks, provide you with all the support you need to write compelling applications for a variety of domains. The problem is, unless you program in Objective-C, Java, or AppleScript, you're stuck. With Jaguar, Apple has introduced a feature called the PerlObjCBridge: a Perl module that allows you to access Cocoa from Perl.

The PerlObjCBridge provides the following functions:

- Enables access to many Objective-C objects from Perl
- Enables Perl scripts to be Cocoa delegates or targets of notifications
- Enables Perl scripts to access Cocoa's Distributed Objects mechanism, letting Perl objects send and receive messages from Objective-C or Perl objects running on different machines

[2] Terminal does not support the KDE-style Terminal, where many shells can be created and used within one window and switched by pressing a key combination.

One limitation of the PerlObjCBridge is its lack of support for accessing Cocoa GUI objects. This means you cannot use it to construct user interfaces for your Perl scripts.[3]

In terms of syntax, Objective-C uses the : to delimit arguments, which is not legal in Perl. Therefore, the _ character is used in its place. To access Objective-C objects and methods from Perl, you use the following forms:

- *Static method* (through the class)—`ClassName->method(...args...)`
- *Instance method* (through the instance)—`$object->method(...args...)`

The following listing shows a few examples of how to use these constructs:

```
# Accessing a method through its class (static method).
$pref = NSMutableDictionary->dictionary();
# Accessing a method through an instance (instance method).
$pref->objectForKey_($key);
```

One of the more powerful features of the PerlObjCBridge is its ability to register Perl objects as recipients of notifications from Cocoa frameworks.[4] For example, the PerlObjCBridge automatically provides the *stubs,* or Objective-C objects that act as proxies for Perl objects. If you have a Perl object

```
package Foo;

sub new { ... }
sub aCallBack { ... }
```

you register `Foo` objects to receive `NSNotification` messages as follows:

```
$foo = new Foo();

NSNotificationCenter->defaultCenter()
->addObserver_selector_name_object_($foo,
 "aCallBack", "theNotificationName", undef);
```

When the event named `theNotificationName` occurs, Foundation sends the `aCallBack` message to `$foo`. Behind the scenes, PerlObjCBridge automatically creates a `PerlProxy` object to stand in for `$foo` wherever an Objective-C object is expected, such as the observer argument to the `addObserver` method.

Cocoa's Distributed Objects (DO) mechanism enables Cocoa programs to access objects from different programs, possibly running on different machines. You can access DO from the PerlObjCBridge, enabling interprocess messaging between

---

[3] See http://camelbones.sourceforge.net for information about a GUI framework for constructing Cocoa interfaces in Perl.

[4] Example and commentary courtesy of Doug Wiebe.

Perl objects. Basically, you write Perl scripts that run on different machines—or in different address spaces on the same machine—and that send messages to one another. Doing so enables your scripts to communicate with other scripts by directly calling their methods as if they were part of the same program.

Let's look at how to apply this knowledge in a Perl script.

8.4.1 PerlObjCBridge example

These days, Palm devices are everywhere. They are used to track contacts and schedules, enter information into databases, access email and the web, and play games. However, most UNIX systems come with software you can use to handle much of this functionally at little or no cost.

The example program for this section is called pim.pl and uses standard UNIX tools to perform tracking contacts, take notes, generate and view calendars, and even keep a list of quotes. The main UNIX programs used are `cal` and `remind`.[5] The `cal` program displays a text-based calendar. If you have used `remind`, you already know how useful it is. If you have never used it, you are in for a treat. Basically, `remind` is a calendar generator and reminder program with lots of options and uses.[6]

The first step in using `remind` is to create and edit your reminders file, called .reminders, which is located in your home directory. Entries in this file represent calendar events or reminders. Once you add entries to the file, you run the `remind` command to process the file. Depending on the options, `remind` will output everything from a reminder list to a text- or Postscript-formatted calendar. Figure 8.11 shows an example .reminders file and a text-based calendar generated from its entries.

To view or edit your contact list, the Perl-based pim script opens the file in an editor; to edit tasks, it opens the task file in an editor; and to view tasks, it processes the file and prints a formatted version of the tasks list.

The Foundation classes include a particularly useful set of methods: the `writeToFile` and `stringWithContentsOfFile` family of methods. Collectively, these methods let you to take an object, serialize its data to disk, and read the stored data from disk into an object in runtime. When used in conjunction with the `NSMutableDictionary`, a hash data structure, you do not need to deal with formatting or parsing

---

[5] The `remind` program is freely available from http://www.roaringpenguin.com/remind. It does not come with the system, so you will need to download it, compile it, and install it for Mac OS X.

[6] See http://www.linuxjournal.com/article.php?sid=3529 for a very good introduction to `remind`.

```
# The .reminders file.
REM 15 November +1 2002 MSG 4-5 Development meeting
REM 05 November 2002 *7 MSG 4:00-5:00 AI seminar
REM 04 November 2002 *1 until November 06 2002 MSG Out of office
REM 04 November 2002 *2 OMIT Saturday Sunday SKIP UNTIL
November 30 2005 MSG Go to gym
```

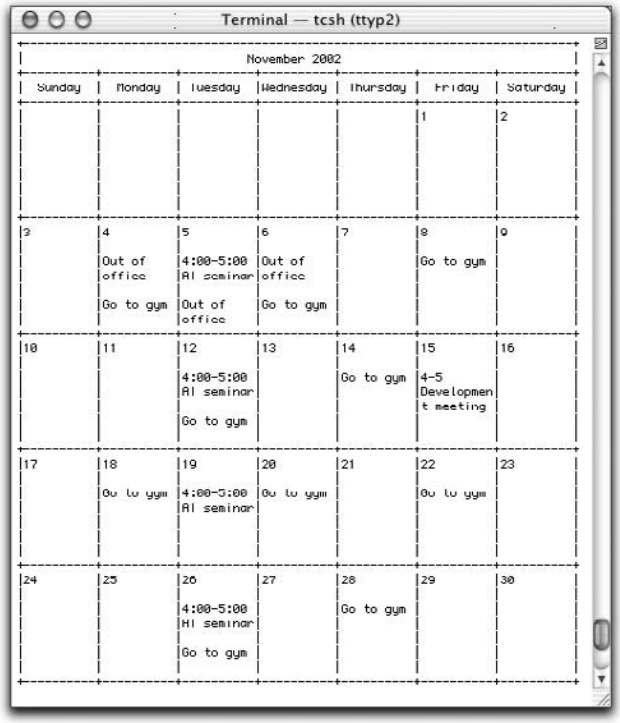

Figure 8.11
An example reminders file and a text-based calendar, generated using `remind -c ~/.reminders`

data; it's all done for you. This feature is particularly attractive and is a strong reason to use Cocoa objects in your Perl scripts. This program uses these features to store and access preference settings.

Application preferences are stored in a preference file, which is a text file formatted as XML. Each key/value pair in the file describes a particular program option. For example, the `editor` keyword is used to look up the editor the script uses to open files (`contacts-file` for the name of the contacts file):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//
DTD PLIST 1.0//EN"
```

```
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>prefs-path</key>
  <string>./</string>
  <key>editor</key>
  <string>/usr/bin/emacs</string>
  <key>viewer</key>
  <string>more</string>
  <key>file-path</key>
  <string>./</string>
  <key>contacts-file</key>
  <string>contacts.txt</string>
  <key>tasks-file</key>
  <string>tasks.txt</string>
  <key>words-file</key>
  <string>word_list.txt</string>
  <key>notebook-file</key>
  <string>notebook.txt</string>
  <key>quotes-file</key>
  <string>quotes.txt</string>
</dict>
</plist>
```

You can initially create this file either programmatically or by hand. To create it programmatically, you can use elements of the following code fragment:

```
my $pref = NSMutableDictionary->dictionary();
setPrefVal("editor", "/usr/bin/emacs");
setPrefVal("contacts-file", "contacts.txt");
writePrefs("pim.prefs");

sub setPrefVal {
  my ($key, $val) = @_;
  $pref->setObject_forKey_($val, $key);
}

sub writePrefs {
  my ($fName) = @_;
  $pref->writeToFile_atomically_($fName, 1);
}
```

When the script starts, it creates a new, empty dictionary object by calling the static method `dictionary` from Foundation's `NSMutableDictionary` class. Next, it populates the dictionary (`readPrefs`) with key/value pairs from the preference file. To do so, it uses the `NSDictionary` static method `dictionaryWithContentsOfFile`. In a single call, it reads the preference file and stores each key/value pair into the dictionary object. This saves you the trouble of creating a new file format and writing code to parse and store the preference values:

```
my $prefs = readPrefs($PREFS_FILE);
sub readPrefs {
  my ($fName) = @_;
  my($dict) = NSDictionary->dictionaryWithContentsOfFile_($fName);
  if (!defined($dict)) {
    logExit("preferences file not read: $PREFS_FILE");
    exit;
  }
  return $dict;
}
```

The rest of the script is quite simple. It goes into an infinite loop in which it displays a menu and handles user selections. To exit the program, press Control-C:

```
for(;;) {
  system("clear");
  print "======================\n";
  print "My PIM\n";
  print "======================\n";
  print "0. Edit preferences file\n";
  print "1. Edit reminders\n";
  print "2. Edit contacts\n";
  print "3. Edit tasks\n";
  print "4. Show tasks\n";
  print "5. Generate calendar (ps)\n";
  print "6. View calendar (txt)\n";
  print "7. Print calendar's";
  print "8. Show system cal\n";
  print "9. Show today's reminders\n";
  print "-------------------\n";
  print "-1. Edit word list\n";
  print "-2. Edit notebook\n";
  print "-3. Edit quotes\n";
  print "-4. View quotes\n";
  print "======================\n";
  print "-> ";
  my $s = <STDIN>;
  chop($s);

  if ($s == 0) {
     system(getPrefVal("editor") . " " . $PREFS_FILE);
  }
  elsif ($s == 1) {
     system(getPrefVal("editor") . " ~/.reminders");
  }
 #…
```

The handling code is also straightforward, as demonstrated by the `if/elsif` block. In both cases, it gets the appropriate value from the dictionary (based on a key) and handles the operation.

Figures 8.12 show the task features.

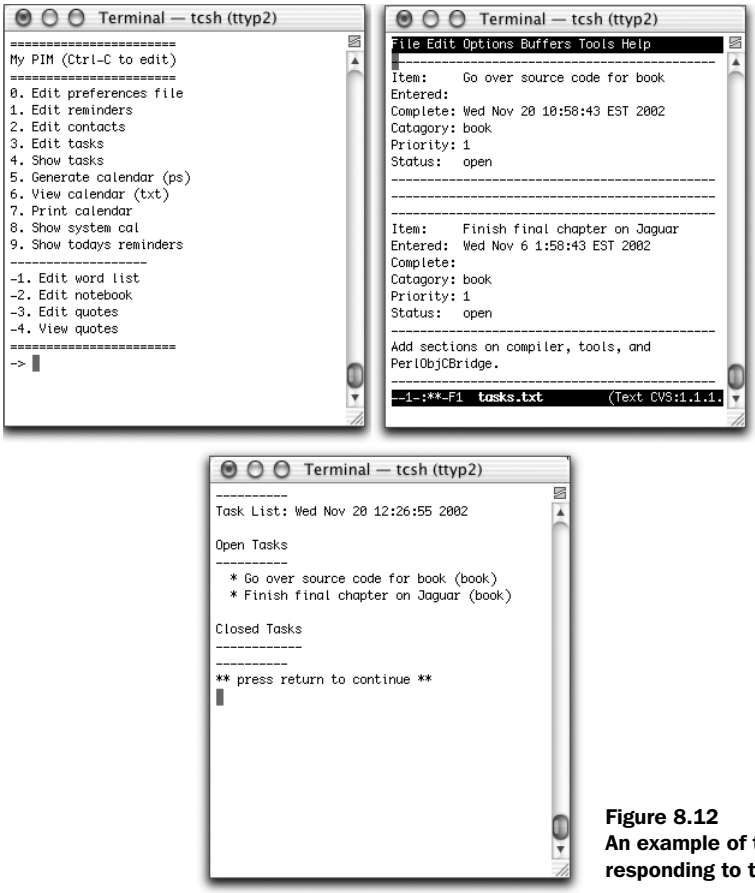

Figure 8.12
An example of the program responding to the Task feature

Overall, this script does the job, but it could be improved. The program could be extended to use Cocoa's DO mechanism. For example, the script could act as a server running on one machine, and you could access it from another machine to access and update information. Additionally, you could write a Cocoa GUI client that displays the information in an interface while the main handling and storage code runs on the server.

Another addition would be to add items to your calendar or task list from your mail inbox. This functionality would let you send yourself email that goes directly into your pim.

Even with these limitations, the program is useful and demonstrates how to use the PerlObjCBridge. The PerlObjCBridge is a good piece of software engineering, providing many more functions than I've discussed here. I hope it will continue

to grow and include more features, including the ability to construct Cocoa GUIs from Perl scripts. If you are a Perl programmer working under Mac OS X, take a serious look at PerlObjCBridge.

8.5 Summary

This chapter has given you a brief look at Jaguar, Apple's first major upgrade to Mac OS X. In many ways, Jaguar extends the current system, but it also adds many new features and enhancements. For UNIX developers, this means `gcc` v3.1, FreeBSD4.4 compatibility, additions to the existing developer tools such as Project Builder and Interface Builder, and new features in the Terminal application. Perl programmers can now take advantage of the Cocoa Foundation through PerlObjC-Bridge, which gives you access to Cocoa objects and features from Perl modules.

Jaguar is a significant step forward for Mac OS X. For UNIX developers and users, there is a lot to like.

A *Getting and installing development tools*

The default load of the Mac OS X system does not include the necessary tools for developing software under Mac OS X (Apple's Developer Tools), including compilers, build tools, Project/Interface Builder, or version control software. To get and install these tools on your system, you need to register for an Apple Developer Connection (ADC) account. Joining ADC is free and entitles you to many programs and services, including:

- Downloading the latest development tools free of charge
- Purchasing the development tools on CD (in case you do not have access to a fast network connection)
- Viewing ADC TV, which broadcasts various developer-related programs including overviews and tutorials on Mac OS X technologies like Cocoa, Darwin, and WebObjects
- Signing up for various programs

To sign up for an ADC account, go to the ADC page (http://www.apple.com/developer) and follow the on-screen instructions. After joining, log in to your new account and download the latest version of the Mac OS X development tools. If you have a high-speed connection, it is best to download the entire archive as opposed to downloading it in segments.

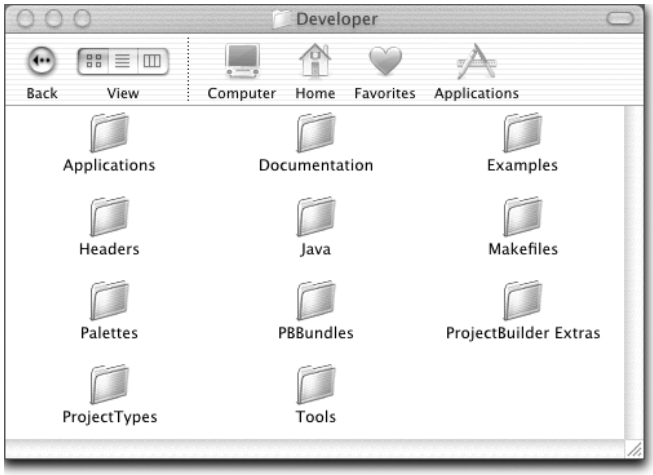

Figure A.1 Once installed, the Apple developer tools are located in /Developer. The folder contains the developer tools, including build tools, documentation, and source code examples.

Once the download is complete, double-click on the archive and follow the on-screen instructions. (You will need administrator privileges to install the development tools.) After you install the tools, a Developer folder will appear at the root level of the file system (/Developer). This folder contains all necessary development tools to build software under Mac OS X (see figure A.1).

B UNIX and Mac OS X command mappings

This appendix lists common UNIX commands and their Mac OS X equivalents. These lists are in no way exhaustive, but they show you how to perform common UNIX operations using Mac OS X procedures.

B.1 Common Mac OS X operations

- To select a single file or folder, single-click on the file or folder you wish to select.
- To select multiple files or folders:
  - Click and drag the mouse over the files and folders you wish to select.
  - Press Command and single-click each file or folder.
- If you have a multibutton mouse, the right mouse button acts like a Control+click.
- To paste a file or folder's path at the command prompt, drag the file or folder from the Finder into the Terminal window.
- To open a directory window from the Terminal, type *open* followed by the directory name at the Terminal prompt.

B.2 UNIX file/directory commands mapped to Mac OS X commands

The mappings in this section show UNIX file and directory commands and their Mac OS X counterparts.

B.2.1 List directory contents: ls

- Maneuver to the target location by double-clicking on each folder in the Finder window.
- Choose a view option from the window's toolbar:
  - Icon
  - List (like `ls -l`)
  - Columns

B.2.2 Copy/move files or folders: cp, mv

Copy files to another location

- Select files and folders (see section B.1) and then:
  - Copy using the mouse by pressing the Option key and dragging the selected items to the destination.
  - Copy using the keyboard by pressing Command+C, maneuvering to the destination, and pressing Command+V.

Copy files to the same location (duplicate)

- Select files and folders (see section B.1) and then:
  - Duplicate using the mouse by right-clicking the selected items (or pressing Control and single-clicking the selected items) and selecting Duplicate from the menu.
  - Duplicate using the keyboard by pressing Command+D.

Move files

- Select files and folders (see section B.1) and then drag the selected items to the destination.

B.2.3 Remove files or folders: rm

- Select files and folders (see section B.1) and then:
  - Drag the selected items to the Trash icon on the Dock.
  - Press Control and single-click on any of the selected items, and select Move To Trash from the menu.

B.2.4 Change directory: cd

- Change using the mouse by double-clicking on the folder you wish to open.
- Change using the keyboard by selecting the folder and pressing Command+O.

B.2.5 Create a new directory: mkdir

- Open the Finder window where you wish to create the directory (folder) and then:
  - Create the directory using the mouse by pressing Control, clicking in the window, and selecting New Folder from the menu. Rename the folder.
  - Create the directory using the keyboard by pressing Command+Shift+N. Single-click on the folder and rename it.

B.2.6 Change file permission and group: chmod, chgrp

- Select files and folders (see section B.1) and then:
  - To use the mouse, right-click the selected items (or press Control and single-click the selected items) and select Get Info from the menu. Change the permission/group.
  - To use the keyboard, right-click the selected items (or press Control and single-click the selected items) and press Command+I. Change the permission/group.

B.2.7 Compare files: diff

- Open the /Developer/Application folder and double-click on the FileMerge program. (See chapter 4 for more information about FileMerge.)

B.2.8 Get the word, line, or byte count: wc

This command has no corresponding Mac OS X GUI program. You can use BBEdit-Lite's Info command (located on the window's toolbar). To get file sizes, use the Finder window's List view (see section B.2.1).

B.2.9 Compress and decompress data: compress, uncompress, tar, gzip, gnuzip, unzip, zcat

- Use StuffIt Expander (located in /Application/Utilities) to extract compressed data.
- Use StuffIt to compress data (http://www.stuffit.com).

B.2.10 Edit text files: emacs, vi

See chapter 4 for more information about Mac OS X editors. Use TextEdit, located in /Applications.

B.2.11 View files: head, tail

- Maneuver to the target location by double-clicking on each folder in the Finder window. Then:
  - Select the Finder window's Column view and select the file you wish to view.
  - To use the mouse, right-click on the file (or press Control and single-click the file), select Get Info from the menu, and select Preview.
  - To use the keyboard, right-click on the file (or press Control and single-click the file), press Command+I, and select Preview.

B.2.12 Find files: find

- Open the /Application folder and double-click on the Sherlock program. Then, select the File Names radio button and enter the search string.

B.3 UNIX communication commands mapped to Mac OS X commands

The mappings in this section show UNIX commands and their Mac OS X equivalents.

B.3.1 OpenSSH: ssh, scp

No corresponding Mac OS X GUI program comes with the system.

B.3.2 Talk to another user: talk, ytalk

No corresponding Mac OS X GUI program comes with the system.

B.4 UNIX process management commands mapped to Mac OS X commands

This section's commands show mappings between UNIX process management commands and their Mac OS X counterparts.

B.4.1 Show system and process usage statistics: top, ps

- Open the /Applications/Utilities folder and double-click on the CPUMonitor or ProcessViewer program.

B.4.2 Terminate a process: kill

- Press Command+Option+Esc, select the process to kill from the list, and click the Force Quit button.
- Open the /Applications/Utilities folder and double-click on the ProcessViewer program. Then, double-click on the process you wish to kill.

C
The precursor of Mac OS X: Mac OS

Before Mac OS X appeared in March 2001, Macintosh computers used another operating system: *Mac OS*.[1] The Mac OS was the primary operating system for the Macintosh line of computers from its appearance in 1984 until the introduction of Mac OS X. The roots of Mac OS go back to the early days of Macintosh computing and exerted a noticeable influence on Mac OS X. For this reason, it's useful to understand some of the spirit and components of Mac OS.[2]

This appendix introduces you to the Macintosh interface and operating system (Mac OS). After reading it, you should have a good understanding of the design goals that led to the Macintosh user interface, the basics of Mac OS, and the underlying system components that compose Mac OS. Against this backdrop, the strengths and weaknesses of the Mac OS emerge, and you can better understand and appreciate the technical developments that led to Mac OS X.

C.1 A tour of the Mac OS interface

The Mac OS user interface is simple and intuitive compared to UNIX desktops and window managers such as GNOME, KDE, CDE, `fvwm`, and WindowMaker. Figure C.1 shows the central user interface metaphor for the Mac OS: the *desktop*.

The Macintosh desktop is a metaphor for a real office desk and provides a display and manipulation surface for information. The *Finder* manages the desktop. The Finder works with the system software to provide users with file management and process invocation functions, and presents and manages the desktop.

Mac OS centralizes and presents several interface elements that users use to interact with the Macintosh system. At the top of the screen is the menu bar, which contains menus enabling users to perform system-related tasks. The system fixes the location of the menu bar so a user cannot move it. This behavior is different than that of the Windows or UNIX environment, where the menu bar appears at the top of each application window. Mac OS programs order menus as follows (left to right):

- The Apple menu is a system-wide menu you can customize. It enables direct and centralized access to commonly used items such as programs, system settings, printers, and network servers.

[1] The official name is MacOS (no space) followed by a version number—MacOS 9. Because I am discussing the system as a whole, I will refer to it as Mac OS.

[2] The Macintosh interface was related to the Apple Lisa, which descended from the Xerox Star.

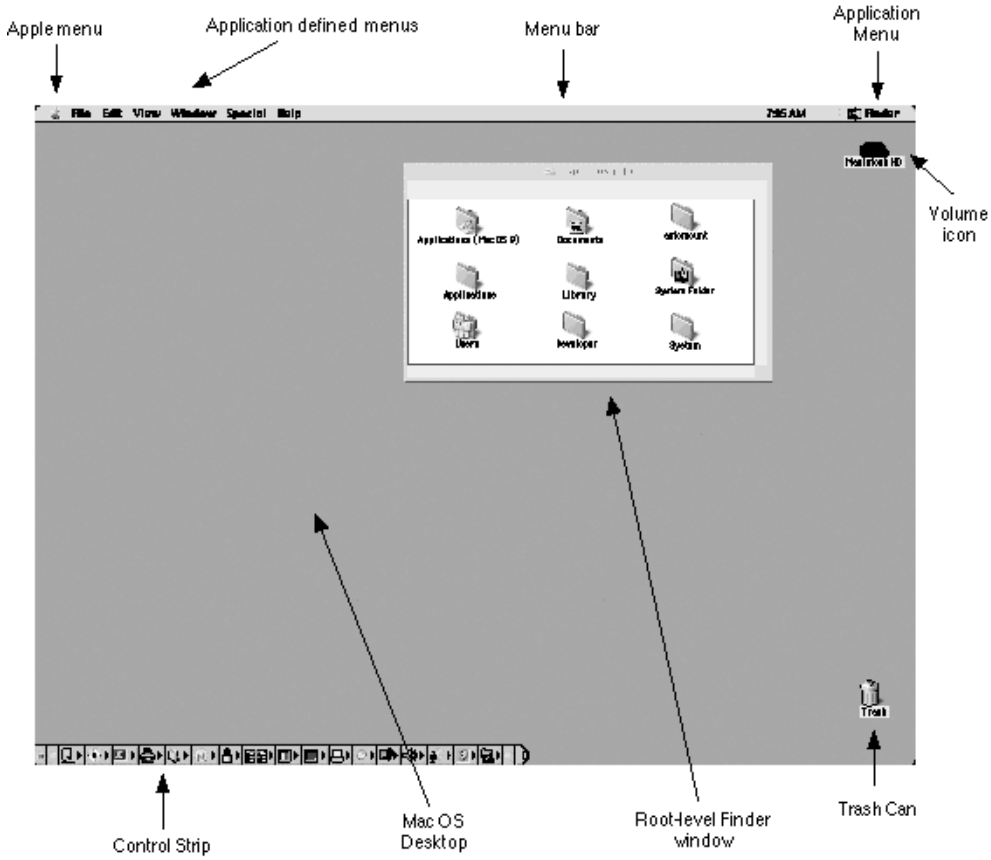

Figure C.1 The Mac OS desktop provides a common workspace for using the system and centralizes many user interface elements, represented by icons.

- The next set of menus is application defined, but should begin with the File and Edit menus and end with an optional Window menu. The system automatically supplies the Help menu. The File menu provides commands for document management, such as opening an existing document, creating a new document, or printing a document. The Edit menu contains commands for editing application documents and sharing applications data via the clipboard. The Window menu holds commands for displays and maneuvering application windows. The Help menu provides users with access to application help.
- At far right on the menu bar is the Application Task menu, which lists all instantiated applications and enables you to change between programs.

The desktop is both aesthetic and functional. You can personalize the environment by choosing different images to display on the desktop. The desktop is a display and manipulation surface for applications, files, aliases (soft links), and mounted file systems and network volumes. In addition, it provides the system with a common focus, or glue, that enables a smooth user experience when moving from application to application.

Formatted disk partitions, or *volumes*, are usually displayed at the upper right on the desktop. At the bottom of the screen is the Control Strip, which provides easy access to applications and system services through a strip of icons that represent each program or service. The Control Strip is optional and can be resized, moved, hidden, or minimized. To launch an application or modify a service's settings, you click on its icon in the Control Strip.

At lower right on screen is the Trash icon, which is a metaphor for a real-world trash can. To delete items such as folders or documents, you toss them into the Trash by dragging them to the trash can icon. When items are in the trash can, its icon changes to reflect this fact. To delete all items from the Trash, choose Empty The Trash from the Finder's Special menu.

C.2 Interacting with the system

The Macintosh interface primarily uses real-world metaphors to represent computer resources. Navigating and using the system is simple because you already use many of these metaphors in your daily life.

On the Macintosh, you organize information in folders (directories). Folders can contain other folders, documents (files), or other items. Icons represent interface elements such as folders, documents, and disks, which you use to maneuver the system and perform operations.

You view and interact with folders and documents through windows. To close a window, click on its close box. To resize a window, click on the window's resize box and, while holding down the mouse button, drag the corner of the window to its new location. The zoom box toggles between displaying all items in the window and the original size. To maneuver the window, click on one of the scroll arrows, the scrollbar itself, or the scroll box (also called the *thumb*), drag it to the new position, and release the mouse button.

To maneuver through the file system, you typically use the mouse to double-click on icons representing folders until you reach the desired location. Then, you can perform these tasks:

- To launch an application, double-click on the icon for the program.
- To move a folder or document, click on the icon, drag it to the destination (perhaps the desktop or another folder), and release the mouse button.
- To copy an item, do the same thing as in the move operation, but hold down the Option key.
- To delete items, click on them and drag them to the Trash.

In all cases, you can select sets of items by holding the Shift key and single-clicking on each item. Each time, you first select what you wish to work with; then you do something with it.

The Mac OS interface contains many more items and features, but these fundamentals are enough for you to understand the basics of the interface and its navigation. For more information, see the Macintosh Human Interface Guidelines and Mac OS 8 Human Interface Guidelines.

C.3 Mac OS system components

This section introduces you to the main technical features of Mac OS, from its initial release in 1984 through the present. These features are contrasted with similar features of the UNIX operating system.

The heart of any computer operating system is the kernel. Broadly speaking, you can view Mac OS as a collection of cooperating system components, distributing system services between system components. In contrast, Mac OS X is based on a microkernel design.

An important element of the Macintosh system, although not technically part of the traditional notion of an operating system, is the *User Interface Toolbox*, or simply the *Toolbox*. The Toolbox, which was originally located in the ROM but now resides on disk, is a set of functions that programs access to construct the graphical elements of a program and interact with core system components. The Toolbox gives the Macintosh its unique appearance and feel. It defines and implements routines that handle user interface components like dialog boxes, windows, and menus; it also provides foundational drawing routines, as well as sound, printing, memory management, and file services. By providing a fairly complete set of controls, the Toolbox encouraged all developers to use the same controls, thus making applications similar in look and function. Each group of related routines are arranged within Managers. For example, memory management functions are part of the Memory Manager; file functions are part of the File Manager.

Figure C.2 shows a high-level view of the Macintosh system.

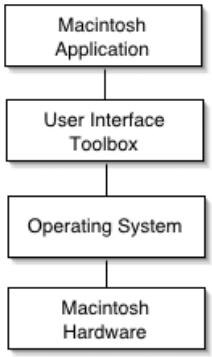

Figure C.2
The Mac OS system consists of two primary system-level software components: the operating system and the User Interface Toolbox.

C.3.1 System file and Finder

Two important files reside on every Macintosh startup disk: the System file and the Finder. The System file contains components necessary to run the computer. If the startup disk does not contain a System file, it will not boot. Don't confuse the System file with the kernel—they are two different things. Similarly, the System file contains resources such as fonts, which are not part of a classic kernel.

The Finder is a program that runs at startup; it provides file-management and process-invocation functions, and presents the user with the familiar Macintosh desktop. Later versions of Mac OS X have a multithreaded Finder, allowing users to perform other tasks as the Finder executes operations such as copying files.

The System file and Finder are located within the System Folder. The System Folder collects programs and resources that the operating system needs to run the system.

C.3.2 Process scheduling

Process scheduling under the Macintosh has evolved significantly over the years. In the early days of the Macintosh, the system was very much a single-process computer. Switching between programs required you to quit the current program (which returned you to the Finder) and launch the other program.

The program Switcher (by Andy Hertzfeld) and Apple's MultiFinder added some process management extensions to the Macintosh. Switcher worked by dividing the Macintosh's memory into several sections, each of which could hold a single program. Switcher permitted users to load a single program into a memory section and switch between the programs, without quitting the current program and returning to the Finder. However, all programs shared the same screen, so it was not possible to view more than one program concurrently—Switcher was non-multitasking.

Apple's MultiFinder, released in 1987, was a *cooperative multitasking environment* that allowed all loaded programs to be viewed on the same screen. The functionality of MultiFinder was later migrated to Apple's System 7. Still, it was up to the user, not the operating system, to switch between programs.

Cooperative multitasking is implemented as follows. When a user runs a program by double-clicking on its icon in the Finder, the operating system loads the program into memory and schedules it for execution on the CPU. The program runs only when the currently running program surrenders the CPU. It is the responsibility of each program—not the operating system—to occasionally hand over the CPU to allow other programs to run. This implementation is considerably better than previous scheduling methods, but it is not optimal because one rogue program can monopolize the CPU and disallow other program from running.[3]

Contrast this approach with the way UNIX traditionally schedules processes. The UNIX process scheduler divides CPU time into time slices, assigning each process a quantum of CPU time. If the running process has not terminated by the end of its quantum, the operating system performs a process switch by preempting the running process and activating the next. The priority of a process is usually taken into account when choosing the next process to run. Process scheduling can also be altered at the user level through the `nice` and `renice` commands, which let you change the current scheduling priority of a process.

C.3.3 Memory management

The Mac OS organizes memory into system and application partitions. Figure C.3 provides an overview of the Mac OS memory layout.

System components exclusively occupy the system partition; they include system software, extensions loaded at boot time, and device drivers. The lowest area of memory is reserved for the system's global variables, which include the current application, the system uptime, and the height of the menu bar. Programs typically access this information through Toolbox calls; the Mac OS does not restrict direct access.

The next area of memory is the system heap. It holds operating system code and data structures.

---

[3] In the Macintosh world, programs are user driven. In a program's main event loop, it calls the `WaitNextEvent` function, which waits for an event to happen and switches cooperatively to other applications. This is done transparently to the running application.

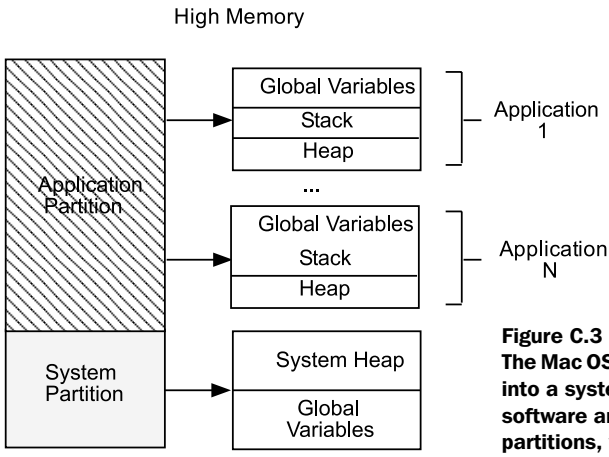

Figure C.3
The Mac OS memory model is primarily organized into a system partition, which holds system software and data structures, and application partitions, which hold application programs.

Above the system heap is a partition of memory for loading programs. When a program is loaded, it's allocated a chunk of memory from the application partition based on the application's SIZE resource. On the Macintosh, a resource holds a specific piece of information for an application. (Resources are discussed in more detail in section C.3.7.) The SIZE resource tells the operating system the program's preferred and minimum memory requirements. Each application partition is divided into segments that contain items such as the program heap, stack, and global variables. On the Macintosh (68k architecture), stacks grow toward low memory and heaps grow toward high memory. As with other operating systems, the heap is used mainly for allocating memory.

As a program runs, it typically allocates and deallocates memory from its application heap. Over time, the application's heap can become fragmented, meaning that unused memory areas are not contiguous; therefore memory requests that exceed the size of the largest unused memory area will be rejected, even though cumulatively enough free memory exists. To address this issue, the Memory Manager performs periodic heap compaction and purging in an attempt to relocate smaller chunks of memory into contiguous blocks. As you can imagine, moving memory within the heap at runtime can wreak havoc on application programs that rely on the addresses of allocated memory.

Under Mac OS, these semantics are implemented using *handles* and *pointers*. Handles can be moved, but pointers are stuck and cannot be compacted. A handle is a pointer into a *master pointer block* that holds pointers to allocated memory. These pointers can move during heap compaction, which results in stale pointers. The handle stays valid, even though the pointer it points to has changed. You can

lock handles to prevent the master pointers from moving, but doing so can cause heap fragmentation. Typically, a handle is locked if it is referenced during an operation that might move memory.

If a program requires more memory than its application heap provides, it can request memory from the operating system, which attempts to allocate the memory from the system's temporary memory area. Virtual memory was added to the Mac OS with System 7.

Under UNIX, processes perform memory management through language-dependent calls such as `malloc`, `calloc`, `new`, and `delete`. These calls are available on the Macintosh in libraries supplied by compiler venders, but Toolbox-defined memory allocation and deletion calls are preferred because they permit more control over the allocated memory within the heap.

The Mac OS does not enforce memory protection of the system or application partitions. Application programs are free to write to memory outside their address space, such as the system space or within other application partitions, and potentially take down the entire system. This scenario is not possible under UNIX because accessing memory outside a program's address space results in a segment fault and the process dumping core, but not taking down the operating system or other processes with it.

C.3.4 Extending the system through system extensions

Extensions have been part of the Macintosh system since the beginning. An *Extension* is a program called a *code resource* that enables you to extend the functionality of your system. Extensions reside in the Extension Folder (within the System Folder) and are loaded by the system at startup.[4] If you add a new Extension to the system, you must reboot for it to be loaded. Extensions are popular among programmers because they allow infinite opportunity to change a system's behavior. On the other hand, they are risky because they are challenging to write correctly. Many of the Macintosh's most creative programs and hacks are Extensions.

Some Extensions work by altering and accessing the Mac OS trap dispatch table.[5] At startup, the system creates two trap dispatch tables in low memory: an operating system trap dispatch table and a Toolbox trap dispatch table. Each table holds a set of addresses pointing to a single operating system or Toolbox routine. The code for these routines is located in either RAM or ROM. When an application

[4] Control panels can also have Extension components.

[5] The true trap dispatch tables have been replaced with TVectors in Power-PC native code.

calls an operating system or Toolbox routine, a trap (exception) is generated. The trap dispatcher looks up the call in one of the tables and transfers control to the routine's stored address. Once the trapped routine is complete, control returns to the caller.

Extensions usually work by first patching a trap dispatch table entry to point to the address of the Extension code instead of the stored routine. When the patched routine is called, control transfers to the Extension and its code is executed rather than the original routine's code. After the Extension code has executed, the original code needs to then be chained and executed. This process sounds simple, but in practice there are various implementation techniques and subtle details to get right. If you make a simple mistake, you can cause unexpected system behavior or, worse, take down the entire operating system.

C.3.5 Interapplication communication (IAC)

The Macintosh implements interprocess communication (IPC)—called interapplication communication (IAC) on the Macintosh—through copy and paste operations using the clipboard, AppleEvents, or the Program-to-Program Communications (PPC) Toolbox.

Copy and paste

The simplest technique for sharing data between programs is copy and paste. A user first selects data from a document and places it on the clipboard through the cut or copy command. Next, the user switches to a different program and uses the paste command to insert the data into the program's document.

AppleEvents

The most popular IAC method is AppleEvents. An AppleEvent is a message whose format is specified by the AppleEvent Interprocess Messaging Protocol. This protocol facilitates sharing data and services between applications. A program that supports AppleEvents is called an AppleEvent-enabled application.

AppleEvents enable applications to extend their functionality with the services of other applications and to share their own operations with others. A common way to interact with AppleEvent-enabled applications is through AppleScript. AppleScript is a high-level scripting language from Apple that sends AppleEvents to applications and system services.

PPC Toolbox

The final IAC mechanism is the Program-to-Program Communications (PPC) Toolbox. An application can use it to send blocks of data to other applications and to receive the data other applications have sent to it. The applications can be on the same system or can reside on a different machine on the network. To use PPC services, both participating applications must be running and both must open a port for communication.

UNIX IPC

UNIX supports IPC through mechanisms such as files and locks, pipes, Berkley sockets and System V derived message queues, semaphores, and shared memory. There is no UNIX system-level IPC mechanism like AppleEvents. Processes send messages over sockets or pipes, but the syntax and semantics of the messages are defined by the individual application programmer.

C.3.6 File system

The Macintosh's original file system—Macintosh File System (MFS)—was flat. In 1985, Hierarchical File System (HFS) replaced MFS. System 8.1 introduced HFS+, which contains many extensions to HFS. Among other things, it increased the number of files possible by adding more allocation blocks, and added support for longer filenames and international filenames.

C.3.7 Macintosh files

One of the unique innovations of the Macintosh team was the design and structure of a Macintosh file. Macintosh files are organized into two components called *forks*:

- The *data fork* is composed of the data component of the file.
- The *resource fork* contains elements called *resources* such as strings, sounds, icons, and runtime memory requirements.

This separation minimizes coupling between related program components and, in some cases, eliminates the need for recompiling a program if a resource element changes. For example, imagine you are developing an application and are implementing error-handling routines. Commonly, programmers hardcode error strings into the application code. If you decide to change an error message, you must recompile the program. File forks change this scenario. Instead, you add error strings to the resource fork within the string resource. Now, if an error message changes, it is independent of the program code—no recompilation is required. Figure C.4 shows a typical resource fork for an application.

M

Mac OS 5, 14, 310
Mac OS 8 Human Interface Guidelines 313
Mac OS 9 23, 80
Mac OS X 5
  architecture 13
  interaction of layers 128
Mac OS X architecture
  compared to Mac OS 14
Mac OS X Server 5
Mach 16–17, 36
Mach kernel 16
  and threads 152
mach_init 36, 44
machine language 324
Macintosh File System (MFS) 319
Macintosh Human Interface Guidelines 7, 313
Macintosh Programmer's Workbench (MPW) 59, 283
Macintosh user interface 6
Macintosh vs. UNIX development 111
MacMAME 23
MacTCP 321
Mail 12, 209, 280
main 139
MainMenu.nib 65
mainWindow 225
make 37, 122
make install 37
makefile 58
malloc 135, 139, 317
malloc allocations
  displaying 165
malloc/free 189
malloc_history 165
  example 166
malloc-allocated buffers
  listing 165
MallocDebug 135–136, 138, 140, 143, 149
  correcting memory-related errors 143
  debugging command-line applications 143
  finding memory-related errors 143
  options 138
MallocStackLogging 166
MallocStackLoggingNoCompact 166
man 32
man pages 32
Managers 313
MANPATH 32
master pointer block 316
memory
  accessing outside address space 317
  allocating 316
  deallocating 183
memory allocation
  checking 135
  collecting 143
  stepping through 143
  viewing 143
memory allocator 135
memory error
  debugging 136
memory leak 140
  detecting 163
  finding with MallocDebug 140
memory management 188, 315, 317
  C 189
  Cocoa 190
  garbage collection 189
  Objective-C 190
Memory Management Utilities 21
Memory Manager 313, 316
memory object
  displaying 165
memory protection 17
Memory Viewer Panel 140
memory-related errors
  detecting 135
menu bar 8, 310
menu item
  adding 103
  changing text 103
  deleting 103
message 177
  routing to views 197
  routing to windows 197
message connections
  setting up 219
messaging 17
  setting up 219
MetaCard 269
method 174, 176
  adding to classes using categories 178
Metrowerks 60
Michigan Terminal System (MTS) 328
microkernel 14
microkernel design 313
Minix 329
mkdir 305
modal dialog box 11
model 194
modeless dialog box 11
Model-View-Controller (MVC) 194, 208
monolithic kernel 14, 17
  scheduling 152
more 53, 112
Moriarity 234, 237
moving files or folders 305
MRJAppBuilder 134
MULTICS 29, 327
MultiFinder 314–315
multiple inheritance 175, 178, 180
MULTiplexed Information and Computing Service (MULTICS) 327
multiprogramming 326
multithreaded application
  debugging 152
  writing 151
multithreading 150
  description 151
mutable string 182
mutex 151
mv 305
MyTask 227, 230–231

N

NDDocument 78, 80
nedit 113, 116
NetInfo Manager 35
Network domain 39
Network Kernel Extension (NFS) 20
networking 16, 19, 321
  services 18
new 317
New File 81

NeXT 24, 101, 182
NeXTSTEP 5, 24
NFS (network file system) 18
Nib file 79, 101–103, 219
nibtool 102
nice 45, 315
non-freed memory allocation
  detecting 165
non-interactive editor 112
noResponderFor 199
nroff 32
NSApp 225
NSApplication 192, 198–199
NSButton 229
NSDictionary 184, 186, 297
NSFileHandle 233
NSFileHandleReadCompletion
    Notification 238
NSMutableDictionary 186, 208,
    221, 295, 297
NSMutableString 182, 222
NSNotificationCenter 238
NSObject 178
NSOffState 229
NSOnState 229
NSOpenPanel 230
NSPipe 233
NSSavePanel 230
NSString 182–183, 231
NSTask 231
  launching subtask 231
NSTraceEvents 200
NSView 198
NSWindow 198
Number 255
nvi 116

O

object 174
  type checking 180
object persistence 180
ObjectAlloc 143
  playback feature 143
Objective-C 173
  background 173
  categories 178
  classes 175
  data members 176
  distributed objects 181
  main features 174
  messages 177
  methods 176
  object persistence 180
  protocols 180
  why learn 181
object-oriented
  terminology 174
Omni Group 209
Online Help menu 241
O-notation 185
Open Darwin 54
Open Source definition 334
open source movement 333–334
Open Systems Interconnection
    (OSI) model 321
Open Transport 21, 321
  API 321
OpenBSD 329
OpenGL 22
openPanel 230–231
OpenSSH 307
OpenStep 25
operating system
  communication between
      layers 17
optimization
  common subexpression
      elimination 282
  dead code elimination 282
  profile driven optimization 282
optimization level 96
optimization techniques 281
organizing scripts 251
osaxen 253
outlet 105, 216
  connecting 216–217
  connecting to interface 105

P

PackageMaker 144
page faults
  reporting 160
PAGER 32
Palette window 64
parent class 175
parent process 151
pasting path at command
    prompt 304
pathname delimiter 43
  colon vs. forward slash 43
pbxbuild 97
PDF 22
PEF container 143
PEF Viewer 143
performance analysis 148
performance execution
    profile 97
performance statistics 68
  acquiring 166
Perl 48–49, 53, 61, 112, 201,
    247, 264
  CPAN 252
PerlObjCBridge 202, 280, 293
  example 295
  registering Perl objects as
      notification recipients 294
–pg flag 68
Photoshop 239
pi 256
pipe 48, 319, 335
Pixie 144
playlist 264
plotting 277
plug and play 18
plug-in 81
  building 144
pointer 316
Portable Distributed Objects
    (PDO) 181
portmap 36
POSIX 18, 280
POSIX thread library 151
POST 36
Power On Self Test 36
PowerPC (PPC) 60
PowerPlant 60
preemptive multitasking 14,
    17, 156
preference file 296
Preference Pane 81
preference settings
  accessing and storing 296
private 176
process 152
  getting status
      information 156
  profiling 166
  terminating 307
process automation 246
process identifier 45
process management 45

process scheduling 14, 17, 314–315
process status information 156
process usage statistics 307
processor
  management 17
ProcessViewer 45, 47
profile driven optimization 282
profile-directed
    optimization 281
profiling 147
profiling code 68, 97
program output
  displaying 74
program performance
    analysis 97
programs
  Apple-event enabled 249
  installing from packages 37
  installing from source
      code 37
Program-to-Program Communications (PPC)
    Toolbox 318–319
project
  placing under version
      control 85
Project Builder 61, 67, 144, 283
  Active Target menu 70
  adding files to a project 81
  AppleScript Studio 128
  build 69
  build and debug
      command 69
  build style 67
  Build tab 74
  build tools 68
  build, link options 91
  clean active target 70
  command-line tools 97
  Contents pane 71
    Bookmarks view 73
    Breakpoints view 74
    Classes view 71
    Executable view 74
    Files view 71
    Target view 73
  creating a new project 78
  creating build styles 87
  CVS 87
  debug 69
  Debug pane 74
  debugging 287
  debugging commands 70
  displaying build output 91
  editor 75, 122
    check syntax 77
    indentation 124
    Syntax Coloring 123
    Text Editing option 123
  editor preferences 90
  emacs support 90
  enabling debugging 97
  event tracing 200
  example 62
  external editors 127
  find 74
  gcc 3.1 283
  gcc optimization settings 96
  inline scripting 283
  interface 69
  pbxbuild 97
  preferences 90
  profiling code 97
  project templates 209
  project types 78
    application 78
    bundle 80
    Empty Project 78
    Framework 80
    Kernel Extension 80
    Pure Java 80
    Standard Apple Plug-ins 81
    Tool 81
  Run pane 74
  script menu 286
  semantic code analysis 99
  setting link options 97
  status bar 78
  target 67
  target editor 286
  target, build style example 67
  toolbar 69
  version control 119
  warning flags 100
project template 209
PropertyListEditor 144–145
propriety list 145
  storing application
      parameters 145
protected 176
protocol 180
ps 45, 112, 136, 156, 158, 271, 274, 307
Pslint 135, 189
pthreads 151
public 176
punch card 324
Pure Java 80
Purify 189
pw 284
Python 48–49, 247, 264

Q

Quartz 22, 51, 146, 197
Quartz Debug 146
  example 147
Quartz Extreme 281
QuickDraw 22, 320
QuickTime 22, 281

R

rc scripts 36
RCS 53, 83, 117
  setting up 120
RCS file 120
Real 255
Record 255
reference counting 190
  example 190
regression testing 126
regular expression search 74
release 190–191
remind 295
Remote Method Invocation
    (RMI) 181
Remote Procedure Calls
    (RPC) 181
removing files or folders 305
Rendezvous 281
renice 45, 315
repeat 256
reset 228
resource 316, 319
resource fork 123, 155, 319
Resource Manager 79
responder chain 198–199
retain 190
return 256
Revision Control System (RCS) 83
Rexx 247

Rhapsody 5
rm 305
root 33
root directory 39
Ruby 202, 247, 264
RubyCocoa 202
run levels 44
  Solaris 44
run modes
  Mac OS X 44
  RedHat Linux 44
running operation
  stopping 234
runTask 227, 232
runtime memory
  tracking usage 135
runtime performance
    analysis 148
runtime system
  Cocoa 182

S

sample 166
  example 166
Sampler 148, 161, 166
saveData 222
savePanel 230
say 262
sc_usage 158
scheduling priority 152
scp 307
screen-mode editor 113
screensaver module 81
script
  debugging 261
  running from a menu 286
  storing 286
scripting addition 253
  installing 253
scripting language 48,
    246–247
  background 48
  choosing 264
scripts
  organizing 251
Scripts folder 286
scroll box 312
search and replace 74
search type
  choosing 74
  regular expression 74
  textual 74
searching documentation 287
security 328
  background 39
  BrickHouse 39
  ipfw 39
security policies 18
sed 61, 112–113
selecting
  multiple files or folders 304
  single file or folder 304
semantic checking 121
semantic code analysis 100
semaphore 151, 319
sendmail 36
serialization 180
setAllowsMultipleSelec-
    tion 231
setCanChooseDirectories 231
setCanChooseFiles 231
setStandardError 233
setStandardOutput 233
setState 229
setValue 222
sh 247
shared memory 319
Sheet 12, 225
  displaying 225
shell 29, 58, 328
  background 29
shell command
  executing 284
shells
  changing 30
Sherlock 280, 281
Simula 173–174
single inheritance 178
single-user mode 44
SIZE resource 316
Smalltalk 173–174
SOAP 261
socket 319
Software Configuration Man-
    agement (SCM) 117
software emulation 23
Software Emulation layer 22
source code
  displaying 75
  editing 75
  printing 124
Source Code Control System
    (SCCS) 117
space 256
Splint 121
  annotation 121
  building 121
spooling 326
sprintf 183
ssh 39, 307
stack 316
Standard Template Library
    (STL) 184
startProcess 237
startup items 36
StartupScript file 286
State design pattern 196
static analysis 135, 173
static code analysis 99, 121
static library
  creating 97
static method 176–177
static methods 230
static profiling 283
statically typed languages 173
status bar 78
stderr 74
StdIO buffer
  viewing contents 74
stdout 74
Step Into 71
Step Out 71
Step Over 71
stepping into code 74
Strategy design pattern 196
stream-mode editor 113
String 255
string preferences 21
stringByExpandingTildeIn-
    Path 223
stringWithContentsOfFile
    183, 295
stringWithCString 183
stub 294
StuffIt 306
StuffIt Expander 306
subroutine 259
subtask
  launching 231
sudo 34
SuperCard 269
superview 199

Swing 80
Switcher 314
symmetric multiprocessing 17
syntax
  checking 77
syslog 36
system call usage statistics 158
system calls
  reporting 160
System domain 40
System file 314
System Folder 314
system heap 315
system logs 45
  BSD location 45
  Console 45
system partition 315
System Preference 34, 81
system usage statistics 307
System V derived message queue 319
System V Release 4 329
SystemStarter 36

T

tab 256
tail 306
talk 307
talking to another user 307
tar 306
target 67–68, 75, 87
target action 195
target settings
  viewing 75
Target/Action design pattern 194
task 18
Tcl 247
TCP 321
TCP Wrapper 39
tcsh 31
TECO 113–114
tell 258
Terminal 25, 29–30, 51, 280
  anti-aliasing of fonts 293
  copying paths to shell 31
  dragging text between windows 293
  splitting window 292
  transparent window 293
  vt100/vt220 emulation 293
Terminal preferences
  setting 289
terminating a process 307
Test Interface 107
Text 255
Text Encoding Conversion Manager 21
text files
  editing 306
TextEdit 271
textual search 74
THINK C 59
Think Class Library 60
THINK Pascal 59
thread activity
  displaying 153
thread library 151
Thread Manager 21
Thread Viewer 150, 152
  example 152
  vs. sc_usage 160
threading 150
threading primitive 151
threads 18, 151–152
  current state 152
  scheduling 17
thumb 312
timeout 258
time-sharing 324, 326, 328
  systems 29
Tkinter 129
tool
  building 144
Tool Command Language (Tcl) 127
Toolbox 24, 313
top 45–46, 53, 112, 136, 156–158, 292, 307
  examples under different OSs 157
trace file 261
trap 318
trap dispatch table 317
trap dispatcher 318
Trash 312
true 256
trueprint 124
try 258
twm 51
type code 320

U

UDP 321
UFS (UNIX file system) 18
uncompress 306
UNIX
  commands under Mac OS X 46
  communication commands 307
  development philosophy 335
  etymology 328
  file/directory commands 304
  kernel 329
  open source software installing 207
  philosophy 111
  process management commands 307
  releases 329
  technical features 329
unreachable code
  removing 282
unzip 306
update 36
user accounts 33
  administrator 33
  creating 34
  enabling root 35
  root 33
User domain 39
user interface 7–8
  Apple menu 10
  application menu 10
  application-defined menus 10
  Aqua 8, 26
  component layout 144
  creating 100
  designing 100, 132
  desktop 8
  dialog boxes 11
  Dock 10
  Drawers 12
  menu bar 8
  testing 107
  window layering 11
userland 20, 112

V

variables
  viewing contents 74
version
  checking out 86
version control 85, 117
  background 117
  choosing 117
  example 117
  geographically distributed team 119
  multiple developers 117
vi 53, 112–113, 116, 306
  terminal vs. GUI version 116
View Layout Rectangles 213
View menu 10
viewing binary files 126
viewing files 306
views 194, 224
  cycling between 77
  updating 228
vim 113, 116
Virtual File System 43
virtual memory 17, 317
  management 328
Virtual PC 23
virtual storage 328
Visitor design pattern 196
volume 312
VSIZE 158

W

warning message 100
wc 48, 306, 335
wget 205
  example 205
  installing 207
WGetParameters 221
window
  adding 103
  closing 312
  layering 11
  management 22
  manager 16
  moving 312
  resizing 312
Window menu 311
word count 306
worksheet 59, 283
writeToFile 183, 295
writeToURL 183

X

X server 113
X Window 51
  installing on Mac OS X 52
  rooted (full screen) 52
  rootless 52
X11 applications 52
XDarwin 52
XEmacs 115
XMethods 261
XML services 21
XML-RPC 261
xxd 126

Y

ytalk 307

Z

zcat 306