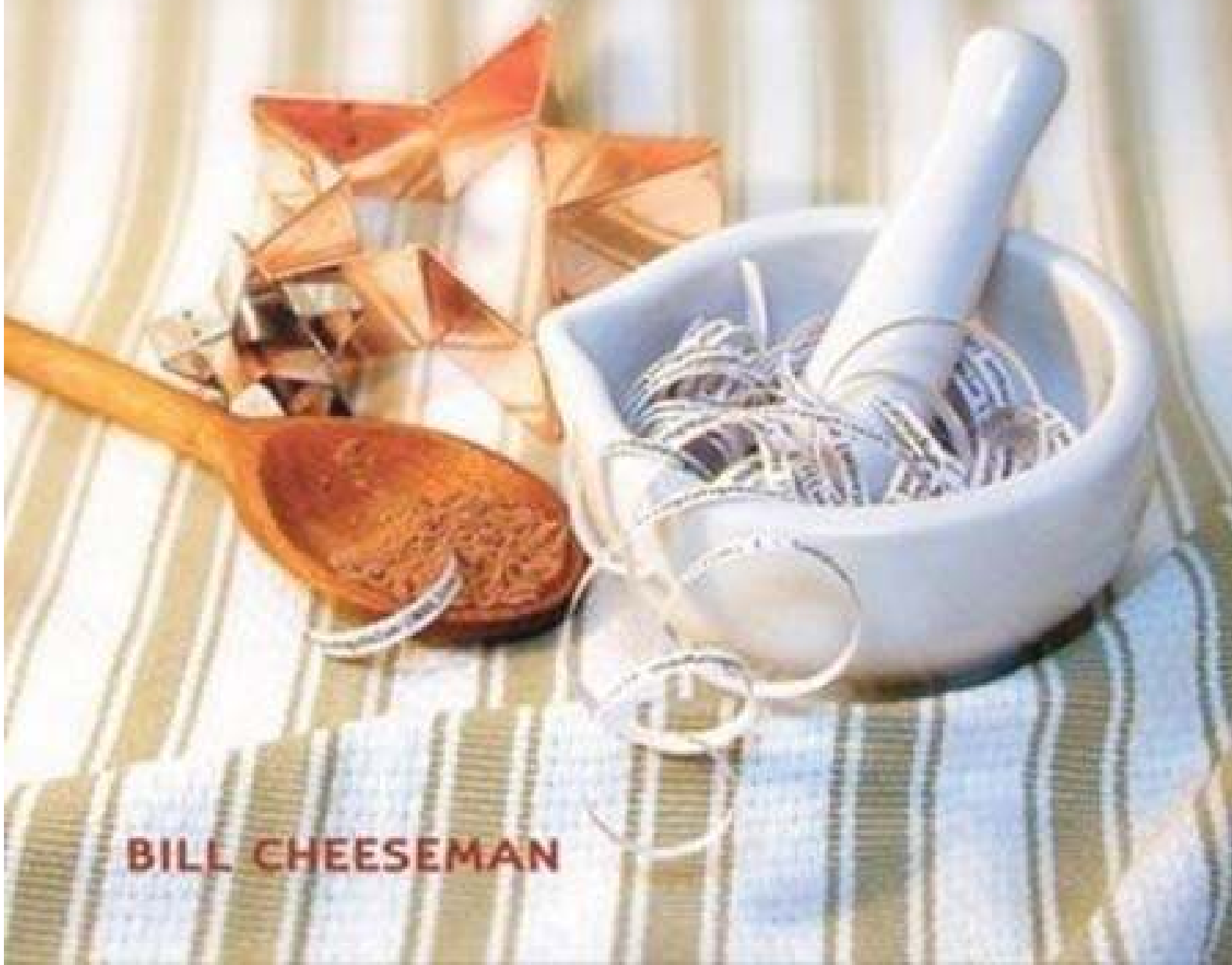


Based on the popular Vermont Recipes Web site,
which helped the new generation of Cocoa developers learn how to write Cocoa programs

Cocoa Recipes for Mac OS X

SECOND EDITION

The Vermont Recipes



BILL CHEESEMAN

Cocoa Recipes for Mac OS X

SECOND EDITION

The Vermont Recipes

BILL CHEESEMAN



Cocoa Recipes for Mac OS X, Second Edition

Bill Cheeseman

Peachpit Press

1249 Eighth Street

Berkeley, CA 94710

510/524-2178

510/524-2221 (fax)

Find us on the Web at: www.peachpit.com

To report errors, please send a note to: errata@peachpit.com

Peachpit Press is a division of Pearson Education.

Copyright © 2010 by William J. Cheeseman

Editor: Rebecca Gulick

Production Coordinator: Myrna Vladoic

Compositor: Debbie Roberti

Copy Editor: Elissa Rabellino

Proofreader: Liz Welch

Technical Reviewer: Michael Tsai

Indexer: Valerie Haynes Perry

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit Press shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Apple, Cocoa, Mac, Macintosh, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. Other product names used in this book may be trademarks of their own respective owners.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN 13: 978-0-321-67041-0

ISBN 10: 0-321-67041-8

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To Mom and Dad.

*A certified public accountant and an electrical engineer
who set me on the right track.*

ACKNOWLEDGMENTS

After six years, I am still indebted to the people I thanked for helping to make the first edition of *Cocoa Recipes for Mac OS X* possible. This, the second edition, builds on the foundation of the first, and they have all continued to provide the support, encouragement, and knowledge without which I could not have added all the new material in the second edition. Thanks again to my family and to all of you at Apple, at Peachpit, at Stepwise, and in the community of Macintosh developers. And special thanks to Michael Tsai for doing a great tech review.

Table of Contents

Introduction	xi
About <i>Vermont Recipes</i>	xii
Why Cocoa?	xiv
Why Objective-C?	xvi
Naming Conventions	xvii
Apple's Cocoa Documentation	xviii
Xcode and Interface Builder	xviii
New Technologies	xx
The Vermont Recipes Application Specification	xxi
Downloading and Installing the Project Files	xxii

Section 1: Objective-C and the Cocoa Frameworks **1**

Ingredients: Language, Frameworks, and Tools	2
Appliances and Utensils	2
Ingredients	3
Serving Suggestions	8

Section 2: Building an Application **9**

Recipe 1: Create the Project Using Xcode	11
Step 1: Create the New Project	11
Step 2: Explore the Project	14
Step 3: Set Xcode Preferences	18
Step 4: Revise the Document's Header and Implementation Files	20
Step 5: Rename the Document's Files	24
Step 6: Edit the Document's Methods	26
Step 7: Create and Revise the Window Controller Files	29
Step 8: Edit the Credits File	33
Step 9: Edit the Info.plist File	35
Step 10: Edit the InfoPlist.strings File	42

Step 11: Create a Localizable.strings File	45
Step 12: Set the Project's Properties and Build Settings	46
Step 13: Build and Run the Application	51
Step 14: Save and Archive the Project	51
Conclusion	52
Recipe 2: Design and Build the GUI Using Interface Builder	53
Step 1: Explore and Revise the Document Window's Nib File	56
Step 2: Add a Toolbar	63
Step 3: Add a Vertical Split View	67
Step 4: Add a Horizontal Split View	78
Step 5: Add a Tab View	79
Step 6: Add a Drawer	80
Step 7: Add a Toolbar Item to Open and Close the Drawer	83
Step 8: Build and Run the Application	87
Step 9: Save and Archive the Project	87
Conclusion	88
Recipe 3: Create a Simple Text Document	89
Step 1: Create the DiaryDocument Class in Xcode	91
Step 2: Save a Snapshot of the Project	94
Step 3: Create the DiaryWindowController Class and Its Nib File in Interface Builder	97
Step 4: Add Scrolling Text Views to the Diary Window	104
Step 5: Create the VRDocument-Controller Class and a New Menu Item	108
Step 6: Add the Diary Document to the Info.plist File	115
Step 7: Read and Write the Diary Document's Text Data	121
Step 8: Configure the Split View Diary Window	133
Step 9: Build and Run the Application	136
Step 10: Save and Archive the Project	136
Conclusion	137
Recipe 4: Add Controls to the Document Window	139
Step 1: Add Controls to the Diary Window	140
Step 2: Implement the Add Entry Push Button	147
Step 3: Implement the Add Tag Push Button	160

Step 4: Validate the Add Tag Push Button	168
Step 5: Implement and Validate the Navigation Buttons	178
Step 6: Implement and Validate the Date Picker	180
Step 7: Implement and Validate the Search Field	186
Step 8: Build and Run the Application	191
Step 9: Save and Archive the Project	191
Conclusion	191
Recipe 5: Configure the Main Menu	193
Step 1: Create the VRApplicationController Class	194
Step 2: Add a Read Me Menu Item to the Help Menu	195
Step 3: Add a Diary Menu to Control the Diary Window	200
Step 4: Add a Diary Tag Search Menu Item to the Find Submenu	202
Step 5: Add a Recipe Info Menu Item to Open the Recipes Window's Drawer	208
Step 6: Build and Run the Application	212
Step 7: Save and Archive the Project	213
Conclusion	213
Recipe 6: Control the Document's Behavior	215
Step 1: Organize the Project's Code	216
Step 2: Limit the Application to a Single Diary Document	221
Step 3: Add Error Handling to the Diary Document	241
Step 4: Prepare Localizable Strings for Internationalization	255
Step 5: Build and Run the Application	257
Step 6: Save and Archive the Project	257
Conclusion	257
Recipe 7: Refine the Document's Usability	259
Step 1: Set the Minimum and Maximum Sizes of the Document Windows ...	260
Step 2: Set the Initial Position and Size of the Document Windows	267
Step 3: Set the Standard Zoom Size of the Document Windows	269
Step 4: Autosave the Position and Size of the Document Windows	274
Step 5: Autosave the Position of the Divider in the Diary Window	282
Step 6: Autosave the Recipes Document's Toolbar Configuration	284
Step 7: Autosave the Diary Document's Contents	285

Step 8: Back Up the Diary Document	297
Step 9: Implement the Revert to Saved Menu Item	298
Step 10: Build and Run the Application	304
Step 11: Save and Archive the Project	305
Conclusion	305
Recipe 8: Polish the Application	307
Step 1: Add a Save As PDF Menu Item	307
Step 2: Use Alternating Show Recipe Info and Hide Recipe Info Menu Items	313
Step 3: Use a Dynamic Add Tag and Tag All Menu Item	316
Step 4: Use a Dynamic Add Tag and Tag All Button	320
Step 5: Use Blocks for Notifications	330
Step 6: Add Help Tags	334
Step 7: Add Accessibility Features	337
Step 8: Provide a Default Diary Document Name	345
Step 9: Add Support for Sudden Termination	350
Step 10: Internationalize the Application's Display Name	351
Step 11: Add Application and Document Icons	353
Step 12: Enable the Application to Run Under Leopard	357
Step 13: Build and Run the Application	364
Step 14: Save and Archive the Project	364
Conclusion	365
Recipe 9: Add Printing Support	367
Step 1: Create a Print Panel Accessory View in Interface Builder	370
Step 2: Create an Accessory View Controller in Xcode	374
Step 3: Add the Accessory View Controller to the Print Panel	381
Step 4: Save Custom Print Settings	389
Step 5: Create a Print View to Print the Document's Content	398
Step 6: Print Custom Headers and Footers	415
Step 7: Implement Print Scaling	421
Step 8: Build and Run the Application	432
Step 9: Save and Archive the Project	435
Conclusion	435

Recipe 10: Add a Preferences Window	437
Step 1: Design and Build a Preferences Window in Interface Builder	438
Step 2: Create a Preferences Window Controller in Xcode	445
Step 3: Configure the General Tab View Item	449
Step 4: Configure the Recipes Tab View Item	459
Step 5: Configure the Chef's Diary Tab View Item	470
Step 6: Build and Run the Application	481
Step 7: Save and Archive the Project	481
Conclusion	482
Recipe 11: Add Apple Help	483
Step 1: Implement an HTML-Based Apple Help Bundle for Snow Leopard	484
Step 2: Add Topic, Task, and Navigation Pages	495
Step 3: Add an AppleScript Link to a Topic Page	502
Step 4: Use the HelpViewer <i>help:</i> Protocol	503
Step 5: Add Keywords and Abstracts	506
Step 6: Add Help Buttons to Alerts, Dialogs, and Panels	509
Step 7: Advanced Help Features	510
Step 8: Implement a Help Book for Leopard and Earlier	511
Step 9: Build and Run the Application	517
Step 10: Save and Archive the Project	517
Conclusion	517
Recipe 12: Add AppleScript Support	519
Step 1: Create a Terminology Dictionary and Add the Standard Suite	520
Step 2: Add the Vermont Recipes Suite and Extend the Application Class With a New Property	525
Step 3: Add a Diary Document Class and a Property in the Application to Access It	537
Step 4: Add the Text Suite and a Document Text Property	542
Step 5: Add a Diary Entry Class and an Element in the Diary Document to Access It	545
Step 6: Add Properties to Get and Set Diary Entry Values	555
Step 7: Add a Current Diary Entry Property to the Document Class	563
Step 8: Support the Make Command for New Diary Entries	564

Step 9: Support the Delete Command for Diary Entries	573
Step 10: Add a Custom Verb-First Command—Sort	574
Step 11: Add Custom Object-First Commands—Encrypt and Decrypt	578
Step 12: Move Along	580
Step 13: Build and Run the Application	581
Step 14: Save and Archive the Project	582
Conclusion	582
Recipe 13: Deploy the Application	585
Step 1: Build the Application for Release	586
Step 2: Test the Application	587
Step 3: Provide Documentation	589
Step 4: Provide User Support	590
Step 5: Distribute the Application	591
Step 6: Promote the Application	595
Conclusion	596
Section 3: Looking Ahead	597
Recipe 14: Add New Technologies	599
Step 1: Switch to Properties	599
Step 2: Switch to Cocoa Bindings	603
Step 3: Switch to Garbage Collection	609
Conclusion	611
Index	613

Introduction

Cocoa Recipes for Mac OS X is a cookbook for developing Macintosh computer applications using the Objective-C programming language in the Mac OS X Cocoa environment. The first edition was written for Mac OS X 10.2 Jaguar. This second edition covers features that are new in Mac OS X 10.6 Snow Leopard as well as the several older versions of the Macintosh operating system that have been released since Jaguar.

To use *Vermont Recipes* to create the Vermont Recipes 2 application that forms its basis, you must install Snow Leopard and the Mac OS X developer tools for Snow Leopard on your computer. Because Snow Leopard does not run on PowerPC Macs, your development computer must be an Intel-based Mac. The finished Vermont Recipes 2 application nevertheless runs both on PowerPC and Intel computers running Mac OS X 10.5 Leopard and on Intel computers running Leopard or Snow Leopard. When Vermont Recipes 2 runs under Leopard, of course, it can't use the new Snow Leopard features described in this book.

The book is subtitled *The Vermont Recipes* because it had its origin on a Web page of that name. The publisher of the original Web edition, Stepwise, was located in Vermont, and the author lives in Vermont. The Vermont Recipes Web page was the first in-depth third-party tutorial about Cocoa, and many of today's Cocoa developers got their start with it.

The book takes a practical, no-nonsense, hands-on, step-by-step approach, walking you through the details of building a Macintosh application from start to finish using the Objective-C programming language and the Cocoa frameworks. It explains in detail what the code is doing and why it works, with only as much theory about the programming language and the frameworks as is needed to understand the approach taken. *Vermont Recipes* places a decided emphasis on getting an application to work correctly as quickly as possible.

About *Vermont Recipes*

Vermont Recipes is distinguished from other Cocoa books and tutorials by its single-minded dedication to creating a complete, integrated, working application. It approaches the job of writing a Cocoa application just as you would approach it in real life, starting where you would start, progressing through the steps that you would follow, and finishing up where you would end. It does not jump from one example to another to illustrate different features of the programming environment. Nor is it organized topically with a multitude of isolated discussions of narrow features of Objective-C and the Cocoa frameworks taken out of context. One advantage of the *Vermont Recipes* approach is that you learn how to organize an application's files and to cope with application-wide interactions that are easily overlooked in a more fragmentary presentation.

The second edition carries this approach even further than the first edition. At the time of the first edition, there was relatively little information available about Cocoa, and the book therefore devoted a lot of space to the details of coding individual user interface elements. With the passage of several years, Apple documentation and third-party books and Web sites have amply covered topics like that. The second edition therefore does not focus on such details but instead covers them in passing as needed to write the *Vermont Recipes* application.

Once you've finished *Vermont Recipes*, you will be able to follow the same sequence of steps to write a full-featured application of your own. Think of *Vermont Recipes* as a detailed cookbook for a magnificent banquet, and enjoy the many other excellent Cocoa programming books that have appeared as appetizers, side dishes, and desserts, as your taste dictates.

The emphasis here is on code. *Vermont Recipes* is not a tutorial on the finer points of Apple's development tools, Xcode and Interface Builder, although the steps necessary to create an application are discussed in more than enough detail to get you through the job. Nor does it explain all of the niceties of Objective-C syntax. Instead, it offers discussions of important features of the language, such as categories and protocols, in the Cocoa context, as they are encountered in the developing application. It is an ordered collection of do-it-yourself recipes—ingredients consisting of commented and organized code snippets, together with instructions for assembling them into a working whole—to guide you through the process of creating classes and subclasses, objects, properties, outlets, actions, and all the other pieces, until you have built a fully functional Cocoa application.

Vermont Recipes is a cookbook, but it doesn't include the kitchen sink. The Cocoa frameworks and related libraries and frameworks contain hundreds of classes, and this book makes no attempt to show you how to use all of them. What it does do is

show you how the most important of them can be used to create a complete, working application with controls of many kinds and other common application features. It also attempts to teach you, by example and explanation, most of the more general concepts and techniques that are used in Cocoa and Objective-C programming. In combination, what you learn from this book should take you a long way toward an enhanced level of understanding, so that you will find it much easier to master other Cocoa classes on your own.

There is no one right way to design or code a Cocoa application. *Vermont Recipes* takes an approach that works, that is relatively easy to learn, that is consistent and therefore easy to maintain and enhance, and that is sufficiently general to adapt easily to a variety of scenarios. It conforms to conventional Cocoa practices and nomenclature. The application you build here is a *document-based application* using Cocoa's Application Kit and Foundation frameworks, which gives it the flexibility and power to serve as a model for the widest variety of applications. Using the AppKit gives you a large part of the normal, expected functionality of any Mac OS X application “for free,” without requiring any coding on your part. Conforming to the conventions of the AppKit also guarantees you the greatest freedom to add Cocoa technologies to your application as they become available, with a minimum of rewriting.

An important advantage of *Vermont Recipes* stems from the fact that I began writing it while I was learning Cocoa myself during the heady days of the first Mac OS X developer previews—through careful study of Apple's initially sparse documentation and sample applications; online assistance from the NeXTstep, OpenStep, Rhapsody, and Cocoa developer community; and trial and error. It therefore covers ground that I know from personal experience would confuse and frustrate a Cocoa beginner if not spelled out in some detail. A Cocoa developer with longer experience might take many issues for granted and therefore neglect to cover them. At the same time, this second edition is based on six additional years of experience writing a number of Cocoa applications, including two commercial applications.

The initial, Web-based version of *Vermont Recipes* was written for the Mac OS X developer previews and Mac OS X 10.0 Cheetah in 2000. The first printed edition was written for Mac OS X 10.2 Jaguar, released in 2002. Apple released Mac OS X 10.6 Snow Leopard in 2009. The code in this second printed edition of the book works with Mac OS X 10.5 Leopard as well as Snow Leopard.

All Macs that have shipped in the last several years include the latest version of the developer tools as of the date of the system-installation DVD that ships with the computer, so you can install them yourself at no extra cost. Free updates become available from Apple periodically. They are also available as part of the retail Mac OS X product, in case you bought your Mac before Snow Leopard and its upgraded developer tools were available.

The recipes in the book target people who have some programming experience. The book assumes only a modest grounding in the C and Objective-C programming languages and at least a little familiarity with object-oriented programming concepts. Both of these can be gained by reading one or two of the many widely available introductory programming texts about C and object-oriented programming, as outlined in Section 1 of this book. No prior experience with Cocoa's predecessors, NeXTstep, OpenStep, and Rhapsody, is necessary, nor is any knowledge of Unix required. You don't need to know Java or C++, and some even say that knowledge of C++ is a hindrance because of the habits you must unlearn.

Only a limited exposure to Objective-C is needed. Objective-C is nothing more than standard C with a few object-oriented extensions. Once you know C's commonly used features, you can learn the Objective-C extensions in a day or two. Having to learn a new programming language is not the obstacle to using Cocoa that some may perceive it to be.

Why Cocoa?

Apple understandably emphasized Carbon development when it initially rolled out Mac OS X, to encourage rapid migration of existing Classic applications from Mac OS 9 to Mac OS X. The Carbon application environment, while complex, allowed those with a longstanding investment in knowledge of the Mac OS toolbox to bring their applications to Mac OS X easily while maintaining compatibility with Mac OS 8 and 9. Apple's strategy paid off, with virtually every major Macintosh application quickly becoming available in native Mac OS X form.

Apple soon positioned Cocoa as the “real” Mac OS X of the future, and it cemented that decision when it announced that it was dropping support for the user interface aspects of Carbon. The company has repeatedly urged developers of new Mac OS X applications to develop them using the Cocoa frameworks. These are mature and powerful application frameworks based on years of NeXTstep and OpenStep experience and the ensuing years of Cocoa. They incorporate virtually all of Mac OS X's functionality and the distinctive appearance of its user interface.

Experienced developers report that the Cocoa frameworks reduce development time by a very substantial factor. This is partly because the frameworks are so complete, giving you a vast amount of functionality without any effort on your part. Apple's Cocoa engineers are fond of demonstrating that you can build a reasonably powerful text editor using the Cocoa frameworks—complete with multiple windows that have scrolling views, a working Fonts panel, copy and paste, drag and drop, unlimited undo and redo, as-you-type spell checking, and other useful features—all simply

by assembling prebuilt features in Interface Builder without having to write a single line of code.

Another factor that makes Cocoa development so efficient is Cocoa's use of powerful design patterns not found in many other environments. For example, a number of the Cocoa classes employ the concept of *delegation*, where the system automatically calls a method of a delegate object when a significant event takes place, such as the user's attempting to close a window. You write the delegate method yourself, or you decide not to implement it at all, which allows you to decide whether and how your application should respond. These hooks, liberally sprinkled throughout the Cocoa frameworks, let you customize application behavior by performing additional actions or vetoing actions that the system proposes to take, depending on conditions that exist at run time. You usually do not have to subclass the built-in Cocoa AppKit classes to gain the benefit of these optional but powerful features. Many of the Cocoa classes also post *notifications*, so that any of your objects can learn of events as they occur elsewhere in the system and deal with them appropriately. It is vital that you learn these and other features of the hundreds of classes that make up the Cocoa frameworks. While they present a substantial learning curve, these and others, like *protocols* and *categories*, facilitate an extraordinarily productive development experience.

Initially, the classes of the Cocoa frameworks fell into two groups, the Application Kit and Foundation, and these two umbrella groups still form the core of the Cocoa frameworks. Foundation focuses on basic data types, system functionality, and other matters having nothing to do with the user interface. The AppKit concentrates on the user interface and other application features such as documents. Over the years since Mac OS X was introduced, additional Objective-C classes have been added to the mix, to the point where developers no longer agree on which of them are properly considered part of Cocoa and which are non-Cocoa classes that happen to be written in Objective-C instead of procedural C. The distinction no longer matters except to purists, because system-level and Carbon APIs written in C, as well as the new Objective-C classes, can be freely intermixed in program code without difficulty. Developers continue to speak of the *Cocoa frameworks* only as convenient shorthand for code that is centered on Foundation and the AppKit and their associated design patterns.

Many of the Foundation classes abstract the operating system, making it easy, for example, to deal with files and networking at a high level without losing access to any of the power of a lower-level approach. Others provide programming features that are always needed, such as object-oriented collection classes and data types. The `NSString` class, for example, is used throughout Cocoa to handle character-based data, bringing automatic support for Unicode text handling, for conversion to and from different text encodings, and for internationalization and localization.

The AppKit provides classes for implementing windows, all manner of user controls, menus, and all of the other features you need to provide a complete user interface. You can subclass all of these to create custom controls and views, if you wish. The AppKit also provides fundamental classes that support writing a fully integrated, document-based application, including `NSApplication`, `NSDocument`, and `NSWindowController`, which you will learn much about in *Vermont Recipes*.

Why Objective-C?

You can use other programming languages, such as Objective-C++, Ruby, Python, and even AppleScript, for Cocoa development, but most Cocoa developers seem to prefer Objective-C. At one time it was common for developers to question whether Objective-C, which uses a runtime system to dispatch messages, was fast enough for many purposes, compared with Carbon and system-level C APIs. These debates about speed have largely evaporated with the advent of faster computers, although there remain legitimate reasons to optimize code for speed in rare instances where profiling demonstrates the need. Objective-C 2.0, introduced recently, includes new features that improve execution speed in certain circumstances, such as *for* loops.

Vermont Recipes is based almost solely on Objective-C. Objective-C is a surprisingly easy language to learn if you already know C, because it is in fact standard ANSI C with a small number of object-oriented additions. Choosing to develop Cocoa applications in Objective-C may therefore be motivated by nothing more than the wish, for C programmers, to avoid the substantial investment of time required to learn a fundamentally new language.

There are more substantial reasons to use Objective-C, however. Objective-C's dynamic object-oriented extensions to standard C are flexible and powerful, making it possible to design applications in ways that are difficult or impossible using more traditional static programming languages such as C++. Dynamic means, among other things, that Objective-C methods are bound at run time under the control of your code, so you don't have to anticipate the details of a user's actions and lock in your responses at compile time. Instead, your code can respond at run time with a level of flexibility that is not available in other languages. It also means that you are able to use introspection, inquiring at run time about the capabilities of any Objective-C object (for example, whether it implements a particular method). You can do such things as assign method selectors to variables and hand them around at run time for execution by name in response to current conditions.

Developers learning Objective-C often report that they have experienced a magical "Aha!" moment, when their understanding of the language jells and wide new horizons of possibility suddenly become visible.

Naming Conventions

Vermont Recipes follows the naming conventions of Objective-C as they have grown up around NeXTstep and its successors. Some of these conventions—particularly the naming of accessor methods—are actually required in order to take advantage of built-in features of the Cocoa development and runtime environments. Others are work habits that have become more or less generally accepted in the Objective-C and Cocoa communities because they make it easier for other developers to read your code. The following are the most common rules.

- Give a method that gets the value of an instance variable the same name as the variable it accesses. For example, method `myName` gets the value of variable `myName`.
- Give a method that sets the value of a variable a name beginning with `set` followed by the name of the variable with an initial capital letter. For example, method `setName:` sets the value of variable `myName`.
- Start method and instance variable names with a lowercase letter; for example, `init`.
- Start class, category, and protocol names with an uppercase letter; for example, `MyDocument`.
- Prefix class names, exported or global variable names, notification names, and defined types with two or three distinctive letters to avoid contaminating the global name space and running into naming conflicts with other software. For example, use `VR` for some of the Vermont Recipes classes and `NS` for most of the Apple Cocoa classes.

For more information about Cocoa naming conventions, read Apple's *Coding Guidelines for Cocoa*, available on your computer in the Xcode Developer Documentation window.

Apple informally reserves to itself the use of a leading underscore character (`_`) when naming private methods and exported functions. Developers who also use this naming convention, as some do, risk unknowingly overriding private methods in Apple's frameworks, with unfortunate consequences. Apple also uses the leading underscore for private instance variables, but the compiler will catch instance variable naming conflicts in your code.

Newcomers to Objective-C should also be aware of the correct way to identify a method. An Objective-C method can be uniquely identified and accurately distinguished from similarly named methods only if its name, all of its parameter labels, and the colons that separate them are included (some methods take no parameters, so they have no colon). Collectively, these compose the method's name or signature.

For example, the `-close` method is different from the `-close:` method. It would be incorrect and misleading to refer to either of these as a *close method*. This is not only an authoring guideline but also a feature of the language. You often use a method's signature in code in ways that are not common in other programming languages, and your code will malfunction if you do not heed this advice.

A leading minus sign (-) or plus sign (+) before a method name distinguishes instance and class methods in their declarations and implementations, but you do not include them when invoking a method in code. The leading sign is sometimes omitted when writing about methods (as the first edition of this book did), but you must be aware of the difference between instance and class methods.

While on the subject of naming conventions, what about the proper capitalization of NeXT and NeXTstep? Yes, the official name of the company where Cocoa's lineage got its start was NeXT Computer, Inc., and the official name of the product was NeXTstep. Although NeXT, and now Apple, registered trademarks in these names as shown here, it also claimed trademarks in the more ordinary forms Next and Next-Step. You see either form, and others, in use today.

Apple's Cocoa Documentation

Throughout this book, I collect references to official Apple documentation and sample code providing additional information about the topic at hand. To make them easier to find, these collections all have the title *Documentation* and the same distinctive appearance. Once you have installed the developer tools on your computer, most of the Apple documentation mentioned in this book can be found on your computer by searching the Xcode Documentation window for the name of the document as given here. The documentation is also available on Apple's Developer Web site at <http://developer.apple.com/mac/library/navigation/index.html>. Occasionally, I also collect third-party articles, books, and blog entries, with information about where to find them.

Xcode and Interface Builder

Cocoa applications are now almost universally written and built using Apple's principal developer tools, Xcode and Interface Builder. Xcode is today's counterpart of the original Mac OS X Integrated Development Environment (IDE), Project Builder. It handles such tasks as editing, compiling, debugging, and linking code, all under one roof.

A Cocoa beginner may perceive Interface Builder to be nothing more than a convenient interactive graphical user interface (GUI) design utility and Xcode to be the tool for building an application. This perception would be inaccurate, but in *Vermont Recipes* you nevertheless learn how to use Xcode to create a new project and to write its code. Interface Builder is mainly used as a utility to design, build, and test the GUI, but not to generate significant amounts of code. Interface Builder's Read Class Files and Reload All Class Files commands may sometimes be used to update the internals of the Interface Builder nib files when outlets and actions have been added, deleted, or modified in the source code. But Interface Builder is rarely used to generate code using its Write Class Files command, and then generally only for prototyping the initial elements of a new class.

You will discover that the nib files that Interface Builder generates are an integral part of a Cocoa application. A nib file embodies an application's user interface more comprehensively than a simple design tool would. Interface Builder allows you, for example, to use intuitive graphical techniques to tell your code which controls are connected to specific instance variables, or outlets, and which methods, or actions, in your code are triggered by specific controls.

A nib file is not a collection of layout templates or generated code to be compiled along with your application code, as is the case with interface design tools for other development systems. It is, in fact, a set of archived objects that a Cocoa application loads, unarchives, connects, and runs. In this way, Interface Builder allows you to write code that is more completely divorced from a specific user interface, and therefore more portable and adaptable to new interfaces. You may be able to use Interface Builder, for example, to alter the user interface of a compiled application even if you don't have access to its source code, and conversely to prototype and test a user interface without compiling an application.

I have not found an authoritative explanation of what the *nib* in the term *nib file* stands for. It is, of course, the file extension used to identify one kind of Interface Builder file, but what does it mean? Even the original *NeXTstep Concepts* book for version 1.0 of NeXTstep, published in 1990, refers to these files only as *interface files* or *.nib files*. Veterans of those days report that nib stands for NeXT Interface Builder.

One final note about these developer tools: Both Xcode and Interface Builder have undergone rapid change over the years, and there is no sign that the pace will slow. Apple adds new features and even radical changes in form and substance with every release of the developer tools. Even the terminology used in these tools changes over time. It is therefore very likely that passages in this book describing how to work with Xcode and Interface Builder in Mac OS X 10.6 will grow increasingly out of date as time goes by. Reading the *Release Notes* and other documentation for each new version of the tools is important for keeping up with these changes.

New Technologies

Objective-C, Cocoa, and Apple's developer tools have come a long way since the beginning. The most notable new technologies, to my mind, are Cocoa Bindings and garbage collection. The transition to a 64-bit architecture, which comes to full fruition in Snow Leopard, is also important. Others might list properties, blocks, Core Data, Core Animation, or other technologies as more important.

The first two of these new technologies, Cocoa Bindings and garbage collection, are optional in most cases. You can develop an application using the old or the new technologies, at your whim, and your users almost certainly won't notice any difference. The third technology I mentioned, the 64-bit architecture, is theoretically optional, but Apple is now giving substantial indications that it will eventually become mandatory. It requires some coding techniques, especially relating to data types, which differ from the old C-based techniques familiar to Cocoa developers.

The advantage conferred by the first two of these new technologies is strictly for your benefit. They greatly reduce the amount of code you must write, and in some ways they simplify your code. There is every reason for you to take advantage of all of these new technologies.

There is a problem, however. You as a developer need to understand the old technologies. With respect to Cocoa Bindings and another new feature, properties, the new technology still uses the old technology under the hood. In fact, there are good reasons to continue to write accessor methods in many circumstances, so you must learn how to do it. With respect to garbage collection, you may encounter edge cases or other situations where you have to write reference-counted code. This is particularly true if, for example, you write a shared framework that must be capable of working with client applications using either memory management model. Although the use of the 64-bit architecture is not really optional, you still need to be able to translate older C-based concepts and code into 64-bit compatible code.

Vermont Recipes therefore starts out using the old technologies, except that it is 64-bit capable throughout. Once you have mastered them, the book shows you how to convert the code you have written up to that point to the new technologies.

The Vermont Recipes Application Specification

The subject of the first edition of this book was a generic application implementing all of the features typically found in many applications and utilities. These included multiple documents and windows; many kinds of controls, menus, tabbed views, and drawers; and standard Macintosh techniques such as drag-and-drop editing. It was not a focused, topical application designed to serve any particular purpose, such as a music notation tool or a checkbook-balancing program. Instead, it served simply as a showcase for common user interface elements, demonstrating not only how to build them but also how they work when completed. The Vermont Recipes 1.0 application itself didn't actually do anything useful.

I am taking a different approach in the second edition. Many of the same user interface features are coded and described—along with many newer features unheard of in the days of Jaguar—but they are repurposed to serve as user interface elements in a real application that does something useful.

The Vermont Recipes 2 application is specified, in broad strokes, as follows: It is a cookbook application for people who cook food instead of code. Its main window is designed to support a Core Data database containing all of the information needed for a large collection of recipes, although this book will not cover the implementation of the database itself. The application also allows specialized documents of two different types to be open simultaneously, one to contain recipes complete with lists of ingredients, required utensils, and cooking directions, and the other to maintain a diary recording ongoing culinary thoughts and experiences. Each kind of document can be saved separately with its own settings using a file format reserved for that type. Window objects showcasing various categories of controls and views present the contents of the database document and the diary document. The diary document, for example, provides a large, scrollable space for typing text, such as chef's notes and tasting experiences. The application incorporates additional features, such as a Help system and support for newer Apple technologies like blocks. In addition, the application is scriptable using AppleScript. In short, if you're planning to create a multidocument, multiwindow application, the Vermont Recipes 2 application provides a usable model for your own work.

This specification will be modified from time to time throughout the book to add new features.

Downloading and Installing the Project Files

You can download the Vermont Recipes project files from the book's Web site (www.peachpit.com/cocoarecipes) to follow along with the book, if you prefer not to type all the code yourself. The files are annotated with references to the recipes that describe them. If you are a nonlinear thinker, you can start with the project files and look up the explanations in the book.

The downloads come for the most part in the form of compressed zip files. There is a separate file for each recipe, containing the code described up to that point in the book. After downloading one of these files, you will find it in your download folder or on your desktop under a name like *Vermont Recipes 2.0.0 - Recipe 09*, followed by the *.zip* file extension. After double-clicking it to decompress it, drag the Vermont Recipes project folder that it contains to the place where you keep your development files (for example, in your home Documents folder), open the folder, and double-click the *Vermont Recipes.xcodeproj* file to open it in Xcode.

You can also download the completed application and run it on your computer to see all the features that are covered in the book, and more. There are two versions of the completed application, one for Leopard and one for Snow Leopard.

SECTION 1

Objective-C and the Cocoa Frameworks

This is a book about writing software for Mac OS X using the Cocoa frameworks. As far as the typical Cocoa developer can tell from looking at the Cocoa headers, the Cocoa frameworks are written entirely in the Objective-C programming language. Likewise, the application that forms the heart of this book is written almost exclusively in Objective-C.

Although Objective-C was inspired by the object-oriented concepts of the Smalltalk language, Objective-C is in fact the C language with a very small amount of object-oriented syntax grafted onto it. Objective-C is a strict superset of C, which means that the Objective-C compiler can compile all proper C programs. You will see as you progress through this book that you must frequently use plain-old C syntax in Objective-C programs.

To use this book, you must already know C as well as Objective-C. You can write Cocoa applications using other languages, such as C++, Ruby, Python, and even AppleScript. However, the headers, the documentation, and almost all of the example code you will rely on as a Cocoa developer are written in Objective-C, and to read and write Objective-C code, you have to know how to read and write C code.

This book will not teach you the ins and outs of C or even Objective-C. For that, you must turn to other books. The book does, however, begin with this short section outlining what you need to learn about C, Objective-C, and the Cocoa frameworks before you get started, and how you can go about acquiring that knowledge with minimum effort.

Ingredients: Language, Frameworks, and Tools

Recipes typically come with a list of ingredients that you must assemble before you can follow the directions for cooking the dish. The list is usually short and to the point. There is no reason to depart from that tradition here.

Appliances and Utensils

Utensils:

1 Intel-based Macintosh computer running Mac OS X 10.6 Snow Leopard

Most of the cookbooks I'm familiar with don't talk much about the appliances and utensils that are required. They aren't likely to mention that you need a refrigerator, a stove, and a blender, although they usually mention more specialized requirements, like a convection oven. They assume that your cupboards and utensil drawers are well stocked with pots and pans, spoons, spatulas, whisks, and so on, although they might mention unusual items like a pressure cooker.

The requirements for Vermont Recipes are both simpler and more specialized than this: You need a computer. Not just any computer, but a Macintosh computer. Not just any Macintosh computer, either, but an Intel-based Macintosh computer. And it must be running Mac OS X 10.6, known as Snow Leopard.

Snow Leopard is the newest version of the Mac OS X operating system. More than some major releases of Mac OS X, Snow Leopard is intended as a major transition. That may sound strange, since Snow Leopard is widely described as refining existing features, not adding new features. Nevertheless, Snow Leopard represents a huge break from the past, in that it won't run on PowerPC-equipped Macintosh computers. The transition to Intel processors was completed some time ago, but Apple has long supported older hardware for a reasonable period after sale. At last, Snow Leopard marks the end of Apple's support for PowerPC. You must have an Intel-based Macintosh computer to use Apple's newest operating system.

Because Snow Leopard is the future and the future is now, Vermont Recipes focuses on writing software for Snow Leopard. You will learn in Recipe 1 that this means you must build the application under Snow Leopard. The Leopard SDK on which the Leopard compiler and linker rely doesn't contain any of the new features introduced in Snow Leopard. Since you can't build the application under Leopard and Leopard was the last version of Mac OS X that runs on PowerPC hardware, you have no choice but to work on an Intel-based Mac.

You can nevertheless write applications that run under Leopard and older versions of Mac OS X, as well as under Snow Leopard. There will be PowerPC computers in the wild for a very long time to come, and the newest, most advanced version of Mac OS X that they will ever be able to run is Mac OS X 10.5 Leopard. I have two of them sitting to my left now, in the shadow of the two Intel-based Macs in front of me. Compared with Leopard, Snow Leopard is a flash in the pan. It will be here for a while and then disappear when Mac OS X 10.7 arrives in a couple of years. But Leopard will remain running over to my left as long as the fans on those PowerPC machines continue to turn. There is more reason to continue to support Leopard than there was to support most has-been operating systems shortly after their demise.

Ingredients

Language

Ingredients:

1 Objective-C 2.0 programming language, with a dash of AppleScript

The Cocoa frameworks expose an Objective-C interface throughout, and Macintosh applications based on the Cocoa frameworks are therefore generally written in Objective-C. You can mix in other languages, such as Objective-C++ and even AppleScript, using the built-in bridges. But Objective-C is the lingua franca of Macintosh Cocoa applications.

Objective-C is often described as a superset of C. This means that it is C—all of C—with a little dollop of icing on top. Literally every feature of C is included in Objective-C. You could write a program entirely in C without using any of the additional features of Objective-C, and it would build and run successfully using Apple's developer tools. As far as I know, you can still even write a Cocoa program using nothing but standard C, if you are willing to get down and dirty with the Objective-C runtime. There was even a time when Objective-C code was precompiled into standard C and then compiled using a standard C compiler.

If you already know something of C, you will be able to pick up Objective-C in no time. The usual estimate is given as a day or two, and my own experience confirms it. This is a fair estimate even if you know only the most common features of C. You will eventually run into Cocoa framework methods that rely on the more arcane features of C, such as the bitwise operators, the address and indirection operators, C arrays, pointer arithmetic, and so on. But you can learn those as you run into them, so don't let a relative lack of familiarity with C deter you.

If you're completely new to C, then you may have at least a little learning to do before you try to grasp Objective-C. This book is not about C, so you should read one of the many good introductory books on the subject. Whether you're a newcomer to C or have a little or a lot of experience with it, you should acquire a copy of what is commonly known as the *white book*: *The C Programming Language*, Second Edition, by Brian W. Kernighan and Dennis M. Ritchie (Prentice Hall, 1988). It's really all you need, both as a teaching tool and a reference. Well, almost all. The ANSI C standard was updated to C99 in 2000 and K&R has not been updated, so you do need to find something more recent. C99 is the default for Xcode in Snow Leopard, and you can—and should—set Xcode to use C99 in Leopard. A good reference that is more up to date is *C: A Reference Manual*, Fifth Edition, by Samuel P. Harbison, III and Guy L. Steele, Jr. (Prentice Hall, 2002).

An important prerequisite for writing Objective-C code is a basic understanding of the principles of object-oriented programming. I described Objective-C as C with “a little dollop of icing on top.” The icing I was referring to is the Objective-C syntax that adds object-oriented features to standard C. Apple's *Object-Oriented Programming with Objective-C* is an excellent introduction, starting with general principles and then showing how they apply in the Objective-C environment in particular.

Apple's *The Objective-C Programming Language* is the official manual for Objective-C, including Objective-C 2.0. It does not teach standard C or object-oriented programming. It describes only the object-oriented extensions to standard C that define the difference between C and Objective-C. See also Apple's article in the Mac Dev Center, *Learning Objective-C: A Primer*.

A good book that teaches C and Objective-C in tandem in the context of object-oriented programming is *Programming in Objective-C 2.0*, Second Edition, by Stephen G. Kochan (Addison-Wesley, 2009).

Objective-C relies on a runtime to dispatch messages, which accounts for its dynamic qualities. You don't need to understand how the Objective-C runtime works in order to write Cocoa programs, but for advanced users it can help you get out of jams, improve speed where profiling shows the need, and perform unusual tasks. Understanding what makes Objective-C a highly dynamic language that defers as many decisions as possible until an application is running will also help you to

better understand common Cocoa design patterns such as delegation. Read Apple's *Objective-C Runtime Programming Guide*, *Objective-C Runtime Reference*, and *Objective-C Runtime Release Notes for Mac OS X v10.5*.

Objective-C 2.0 introduced a number of important new features to Objective-C. It used to be that you could always tell Objective-C code at a glance because of all those distinctive square brackets, sometimes nested seven or eight deep. That is no longer true, because Objective-C 2.0 allows use of dot notation optionally in place of the brackets. Dot notation is hardly the most important new feature in Objective-C 2.0. Like many, I see dot notation as a solution to a problem that didn't exist, so you won't see any Objective-C dot notation in this book.

You will see a number of other new features of Objective-C 2.0 here, so you should read up on it. Many new features in Objective-C 2.0 in addition to dot notation are optional, at least for now, such as properties and garbage collection. They make it much easier to write Objective-C code by eliminating the need for verbose and tedious memory management code and accessor methods. However, you occasionally run into situations where the new techniques cannot be used, so you have to know how to write code the old Objective-C 1.0 way. In this book, Section 2 is written without using these new Objective-C 2.0 features, and then in Section 3 you learn how to change the code to make use of them.

Snow Leopard brought one new language feature that isn't part of Objective-C 2.0. It is a proposed addition to the C language, so it can be used both in C and Objective-C. I am referring to *blocks*. You will see several examples of blocks in this book.

Frameworks

Ingredients:

1 bag of Cocoa frameworks, seasoned with some Carbon and system frameworks

It is impossible to overstate the breadth and power of the Cocoa frameworks. They have been in development continuously for many years, since the days of NeXT. With the AppKit, Foundation, and the several more specialized Objective-C frameworks that have been added through the years, Cocoa provides a comprehensive set of APIs to develop any kind of application you might fancy.

At my first WWDC, in the days of the Mac OS X Developer Preview, Apple passed out a poster diagramming the AppKit and Foundation. I had mine mounted, and I still have it. It's in my closet, however, because it doesn't show even half of the classes that now exist in Cocoa, only a decade later.

When you attend Macintosh programming conferences such as Apple's own annual Worldwide Developers Conference (WWDC), you constantly hear the phrase "for

free.” Cocoa provides this functionality “for free,” and Cocoa provides that functionality “for free.” And it’s true.

One of the ideas behind the Cocoa frameworks, as I see it, is that you should be able to write applications without getting down to the bare metal, where even simple things are really hard to do, but you should be permitted to get down to the bare metal when you have to. The first part of that dichotomy is the “for free” part of Cocoa. You don’t have to write hundreds of lines of code to create a button; you only have to write one line, or not even that if you use Interface Builder. Various kinds of buttons have standard shapes, sizes, positions, colors, and behaviors in Mac OS X, and you shouldn’t have to write the code to create one yourself. The Cocoa frameworks provide it to you “for free.”

The concept of “for free” in Cocoa also applies at a much higher level. The Cocoa text system is a popular example. You can add a word processor to your application in Interface Builder with no code at all—well, maybe a line or two. This includes full Unicode support; multiple rulers to control paragraph styles, tab stops, line spacing, and the like; spelling and grammar checking; substitutions; transformations; and speech. Think of it: A button and a word processor take the same amount of work.

Ironically, all this “for free” power comes at a high price: You have to learn it. This is far more time consuming than the task of learning the Objective-C programming language. The high cost results in part from the sheer size of the frameworks. It also results from the variety of things you can do “for free.” The Cocoa frameworks don’t force you to use one kind of button. A dozen different kinds of buttons are available, along with the means to customize every aspect of any of them.

Offsetting this cost, fortunately, is an extraordinary cohesiveness. The Cocoa frameworks are known for their unflagging dedication to a number of pervasive design patterns. Once you understand these design patterns, you can learn one Cocoa class after another just by glancing at the list of methods declared in each class. The consistency encompasses everything from behavior to naming conventions. Mastering the design patterns makes it much easier to learn unfamiliar classes. You quickly reach a point where you can predict the names of a new class’s methods without even glancing at them, just from the description of what the class does.

A good place to start is Apple’s *Cocoa Fundamentals Guide*. For an in-depth explanation of Cocoa’s design patterns, read *Cocoa Design Patterns*, by Erik M. Buck and Donald A. Yacktman (Addison-Wesley, 2010).

In this book, you learn many parts of the Cocoa frameworks, but you don’t learn all of the frameworks, and you don’t learn everything about any of them. Instead, I walk you through the stages of developing a specific application, pausing at every step to explain what class and which of its methods you’re using. From time to time, I point out alternative strategies and the classes and methods they would require,

and I mention different facilities you could use if you were writing a different kind of application. My hope and belief is that this process of slow, directed progress through a real-world development project will give you a firmer and more concrete education in the Cocoa frameworks than you could get either by short, disparate examples or by systematic theoretical exposition.

Tools

Ingredients:

1 box of developer tools

Virtually everybody these days uses Apple's developer tools to write applications. They are included on the Mac OS X installer disc on every Mac. All you have to do is install them.

This book is not about the developer tools, so I won't say much more about them here. In Recipe 1, I introduce you to Xcode and the basic techniques you use to begin writing code. In Recipe 2, I introduce you to Interface Builder and the ways in which you use it to design and build a user interface.

To learn Xcode, start with *A Tour of Xcode*. It walks you through the Xcode application, including a workflow tutorial, a section describing recommended reading, and a section on how to use the built-in document viewer to find and read Apple's documentation. The *Xcode Workspace Guide* explains in greater detail the Xcode window structure and the different ways you can set up Xcode to facilitate your personal development style. The *Xcode Project Management Guide* explains the various elements of an Xcode project, where you create and keep track of your code and resources. An essential reference is the *Xcode Build Setting Reference*. Keep it by your side while you are setting up a project in order to get the very large number of build settings right. The *Xcode Build System Guide* is a helpful adjunct, explaining where and how you enter build settings and other aspects of the overall Xcode environment. Whether you are a solo developer or working on a large team, you should consider using a source code management system (SCM) to keep track of your code and resources as you write. Refer to the *Xcode Source Management Guide* for details. For debugging, read the *Xcode Debugging Guide*. For performance testing, Apple provides two basic tools, Instruments and Shark. Read about them in the *Instruments User Guide* and the *Shark User Guide*.

For Interface Builder, rely on the *Interface Builder User Guide*.

Finally, Apple even provides guidance on how to package your application for delivery to end users. The *Software Delivery Guide* is a little long in the tooth at this point but still full of helpful advice. The *PackageMaker User Guide* provides somewhat more recent instructions for creating installer packages for use when Apple's traditional drag-install installation technique is not suitable.

Serving Suggestions

You can write an amazing variety of software on Mac OS X using the utensils and ingredients described here. Most of you are probably focused on writing an application, and that is what this book is about.

But you don't have to outfit a different kitchen to write other kinds of software. Apple's developer tools allow you to build libraries, frameworks, plug-ins, hardware drivers, and many other kinds of software, all using the same tools and the same working environment.

If you are working on a product that requires several different pieces, you can configure Apple's developer tools to integrate them into one development environment. At the simplest level, for example, you can easily arrange to use a single build folder that holds the intermediate and final build files for all the pieces of your product. By default, the build folder for each piece of software is in its own project folder, and that is how you set up the Vermont Recipes project in this book. However, in my work as a solo developer, I customarily put everything in a centralized build folder where my frameworks and helper applications are built alongside the main application. This allows me to set up project dependencies so that every build of one piece automatically rebuilds any other pieces on which it depends.

Xcode allows you to take this a big step further, combining multiple subprojects, configurations, and targets in a single omnibus project. In the Vermont Recipes project covered in this book, you create two configurations and three targets.

But enough of utensils and ingredients. It's time to get to work.

SECTION 2

Building an Application

Vermont Recipes is based on a single application, used throughout the book to provide a consistent and familiar foundation for all of the Cocoa features I will discuss. As you proceed through the recipes and explore Cocoa's myriad capabilities, adding new features to the application a step at a time, you will never be in any doubt regarding the underpinnings of a particular task, because you will have built them yourself. By following the linear path traced in the recipes, you will see how to assemble a working, feature-complete application from start to finish. Once you have made it all the way through Vermont Recipes, you will be able to follow a similar process to build a complete application to your own specification.

In Section 2, you build the application to the point where it has almost everything a normal application requires, but the recipes feature won't have a lot of meat on its bones. You will have an About window; a menu bar with all of the standard menus and menu items and a few custom menu items; a split-view window with a toolbar, a tab view, and a drawer but no other content; a working Chef's Diary document with a few controls; the ability to create, save, and reopen documents and to revert to the last saved version of a document; unlimited undo and redo; and double-clickable application and document icons. In addition, you will have a Preferences window, an Apple Help Book, AppleScript support, and accessibility features—and much more. Now would be a good time to review the Vermont Recipes application specification in the Introduction to remind yourself what it is you are about to build.

The application is created in several recipes. In the process of working through them, you will become familiar with the basic operation of the tools used for Cocoa development, Xcode and Interface Builder.

In the first recipe, you create a new project in Xcode, setting up the initial code files, nib files, and other resources, as well as the correct folder structure for your project. You then

turn to Interface Builder, in the second recipe, to begin laying out the basic features of the application's graphical user interface (GUI), and you even generate a little code. In the third recipe, you return to Xcode to finish setting up the project by setting Xcode and Interface Builder preferences and configuring all of the properties and build settings required to make your application work in its intended execution environment. In the fourth recipe, you begin to write the code that drives the user interface and the application's substantive functions. In subsequent recipes, you implement several of the most fundamental features of any useful application. When you've completed Section 2, you will have a working Cocoa application. The recipes feature will be left for you to complete, but the Chef's Diary and all of the application's other features will be complete.

The Vermont Recipes application is a document-based application relying on the Cocoa AppKit. Like most Cocoa document-based applications, it adopts the Model-View-Controller (MVC) paradigm, which originated in the Smalltalk-80 language from which the Objective-C extensions to C were derived. This is mainstream Cocoa application design, embodying the approach recommended by Apple for typical Cocoa applications, and accounting for much of the simplicity and efficiency of Cocoa development.

Create the Project Using Xcode

Xcode is the core of the Mac OS X Integrated Development Environment (IDE). Apple supplies it free of charge with every new Macintosh computer and with the retail Mac OS X operating system. Use it to build Cocoa applications and other software products. Through Xcode, you access the code editor, the debugger, the compiler, the linker, and other tools. You can run these tools separately using the command line in Terminal, and many developers do, but this book focuses on Xcode because of its overwhelming convenience and power.

HIGHLIGHTS:

Creating and setting up an Xcode project

Setting Xcode preferences

Using the Xcode text editor

Creating a Credits file

Editing an Info.plist file

Setting up strings files for localization

The book is based on Mac OS X v10.6 Snow Leopard and Xcode 3.2. Xcode 3.2 requires Snow Leopard, and like most of Apple's Snow Leopard applications, it will not run on a PowerPC computer. The book therefore assumes throughout that you are developing on an Intel-based Macintosh computer running Snow Leopard. Xcode 3.2 can nevertheless build universal applications to run on the 32-bit PowerPC architecture as well as the 32-bit and 64-bit Intel architectures. The Vermont Recipes 2 application will run under Mac OS X v10.5 Leopard as well as Snow Leopard, with some loss of functionality when running under Leopard with respect to new features available only under Snow Leopard.

Step 1: Create the New Project

Starting with Leopard, the Apple developer tools can be installed almost anywhere, and you can even have different versions of the tools installed on one computer. You know where you installed them, and that's where you'll find the Developer folder. By default, it is located at the root level of your startup volume, alongside the standard Applications folder. I find it convenient to put a link to the Developer

folder in my Finder sidebar so that I can open it quickly to get at other developer utilities provided by Apple, and I put Xcode and Interface Builder in the Dock so that I can launch them quickly. They are located in Developer/Applications.

1. Launch Xcode.
2. Create a new project. The “Welcome to Xcode” window contains buttons you can use to create a new project, to follow an introductory tutorial, or to visit the Apple Developer Connection Web site. It also contains a Recent Projects list. The list is empty for the moment, but you can use it later as a convenient entry point to your work when you have several projects in process. Once you know your way around, you can deselect the “Show this window when Xcode launches” checkbox. For now, click the “Create a new Xcode project” button, or do it the traditional way by choosing File > New Project.

Documentation

Document-Based Applications

In this recipe, you create a new project based on Xcode’s built-in document-based application template. The template assumes that you plan to write the simplest form of document-based application, one that creates only a single kind of document and which opens a single window to edit and view its contents.

The Vermont Recipes 2 application does not in fact take this simple form. Instead, it is able to create multiple documents and to open multiple windows. This recipe guides you through the process of changing the template to use Cocoa features supporting multidocument, multiwindow applications. The techniques you learn in this recipe conform to standard Cocoa practices.

Apple’s *Document-Based Applications Overview* is a venerable document covering what you must do to write any kind of document-based application, from the simplest through the moderately complex to the most complex. Apple has kept it up to date through the many versions of Mac OS X that have appeared since it was first created. Read it as you work through this recipe to supplement what you learn here.

The new project you create in this recipe is not just any document-based application but, in fact, a Core Data document-based application. However, you will not implement the Core Data features of the project in this book, because Core Data is an advanced topic. Apple specifically advises Cocoa newcomers not to try to work with Core Data until they have mastered other Cocoa technologies on which it depends. Ignore the Core Data–related files that the template installs in the project. What you learn here is how to write a non–Core Data application. When you’re ready to implement the recipes feature of the application using Core Data, a good resource is Marcus S. Zarra, *Core Data: Apple’s API for Persisting Data on Mac OS X* (Pragmatic Bookshelf, 2009).

3. The New Project window opens. It is a standard iTunes-style window, with a source list on the left listing basic categories and two larger panes on the right presenting additional choices related to the selected category. Using the source list, you choose the kind of product to build, such as a framework or plug-in. Select the default, Application.
4. Using the top pane on the right, you choose the kind of application to build, such as a command-line tool. Select the default, Cocoa Application.
5. Using the bottom pane on the right, you set various options for the selected kind of application. These settings control the method stubs that Xcode inserts automatically when it creates your new project, as well as specialized support files that Xcode creates for some kinds of applications. Behind the scenes, Xcode selects from a number of templates that are installed with the developer tools.

For now, you know from the Vermont Recipes 2 application specification that it will use Core Data for database-style storage. Select both the “Create document-based application” and “Use Core Data for storage” checkboxes.

In addition, select the Include Spotlight Importer checkbox, so that your application will be a good Macintosh citizen by supporting Spotlight searches. Xcode enables this setting when you select the “Use Core Data for storage” checkbox (Figure 1.1).

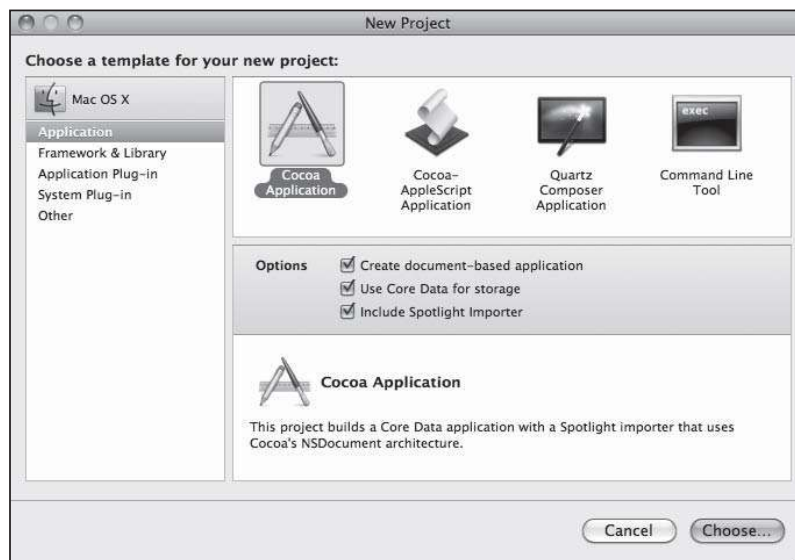


FIGURE 1.1 Setting up Xcode's New Project window.

6. Click the Choose button.
7. In the Save As field of the standard Save panel, enter the name of your new project, **Vermont Recipes**. The default location for the project is your Documents folder, but you can put it almost anywhere. My personal preference is to keep my Xcode projects in subfolders, so that my Documents folder is less cluttered and my development projects are more organized. To follow my practice, click

the disclosure button to expand the Save panel if it is collapsed, and use the New Folder button repeatedly to create a hierarchy of folders in the Documents folder with this path: ~/Documents/Projects/Cocoa/Vermont Recipes 2.0.0.

8. Click the Save button. Xcode creates your new project as a subfolder named Vermont Recipes in the Vermont Recipes 2.0.0 folder, and it opens the Vermont Recipes project window so that you can begin setting it up.

Step 2: Explore the Project

The Xcode project window is another multipane window. Take a quick tour to see what's here. The pane on the left is called the Groups & Files pane. It has the appearance of a Finder-like list of folders and files, but you will quickly come to understand that it is in fact an organizing tool in its own right, mostly unrelated to the way the project files are organized in the Finder. To see the difference, compare the Models and Resources groups in the Xcode project window with the window for the project folder that Xcode just created for you in the Finder.

1. Switch to the Finder and open your new project folder. Start by navigating to the parent folder, which is the new Vermont Recipes 2.0.0 folder if you set up the hierarchy as I have. Nested inside it is the new Vermont Recipes project folder. Open it (**Figure 1.2**).

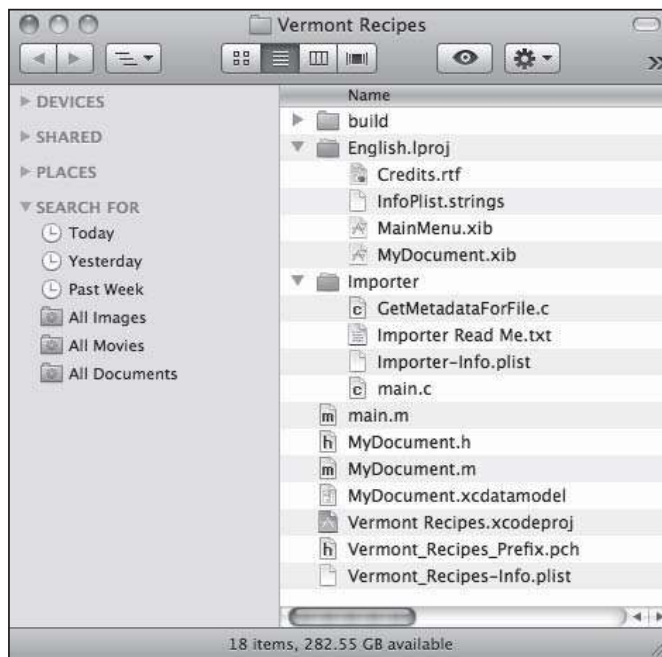


FIGURE 1.2 The Vermont Recipes project folder in the Finder.

For now, assume that every file or folder you place inside the Vermont Recipes project folder is intended to become part of the project itself, so don't delete or

rename anything in it and don't add anything to it using the Finder. You will learn how to add files and folders to the project later. However, you can put anything you like in the enclosing Vermont Recipes 2.0.0 folder, outside the project folder. This is a good place to store information about the project, such as to-do notes, specification outlines, and links to relevant Web sites.

2. Place the Finder's project window alongside the Xcode project window so that you can easily explore and compare them (**Figure 1.3**).

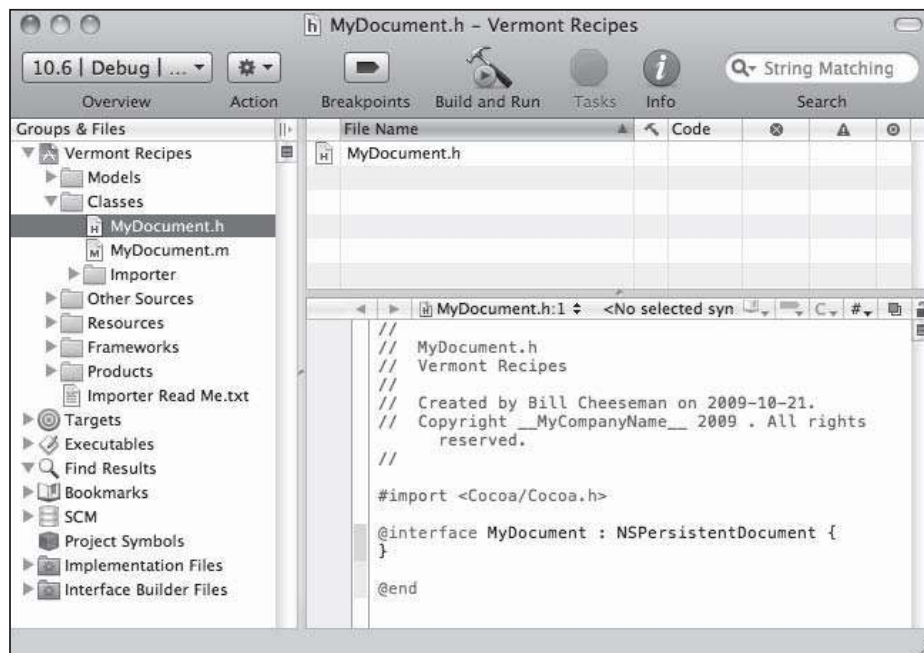


FIGURE 1.3
The Xcode
project
window
after adding
MyDocument.

Expand the Models group in the Xcode project window by clicking its disclosure triangle. It looks like a folder, but there is no folder named Models in the Finder project window. In the Xcode project window, the expanded Models group contains a file named MyDocument.xcdatamodel. In the Finder project window, you see a file by the same name, but it is located at the top level of the window.

This illustrates a characteristic Xcode organizing principle. Most project files are located at the top level of the Finder project window, while the Xcode project window's Groups & Files pane provides organization that you will come to find very convenient. In fact, you can freely create new groups and move them around in the Xcode project window without affecting the Finder locations of included files. The organization of the Groups & Files pane is simply a convenience that you are free to change pretty much any way you like. However, the default organization is based on long experience of the Xcode engineering team with years of developer feedback, so you might want to leave it the way it is for now.

There are some folders in the Finder project window, in addition to the files. One folder is named English.lproj if you are in an English-speaking locale. It is named something else if you are in another locale. When you localize your

application for use with other languages, you or your localization contractor will add multiple `lproj` folders, one for each localization. This book refers throughout to the `English.lproj` folder, but they all work the same way.

3. Now open the Resources group in the Groups & Files pane. Drag the vertical divider to the right if you can't see the full names of the files. You see several files in the Resources group, only one of which, `Vermont_Recipes-Info.plist`, is visible at the top level of the Finder project window.

Where are the others? To find out, expand one of them, `MainMenu.xib`, by clicking its disclosure triangle in the Xcode project window's Groups & Files pane. When the outline expands, you see an item named `English`. This corresponds to the `English.lproj` folder in the Finder project window. Open that folder now in the Finder project window, and you see four more of the files that are listed in the Groups & Files pane's Resources group. These files, such as `Credits.rtf`, are intended to be read by the application's users, so they are language-specific. In a localized application, the expanded Resources group would show an item for each locale, each of them corresponding to an `.lproj` folder in the Finder project window. To edit the English localization of the `InfoPlist.strings` file, for example, you expand the `InfoPlist.strings` entry in the Resources group and double-click the `English` item. Try it. When there is only one localization of a project, you can simply double-click the item without first expanding it to expose the locale.

The last item in the Resources group, `Importer-Info.plist`, corresponds to the file of that name in the Importer folder in the Finder project window.

4. There are two folders in the Finder project window in addition to the `English` folder: the build folder and the Importer folder. By default, Xcode places intermediate, debug, and release files in the build folder when you build your project. The Importer folder contains two code files and an Importer Read Me text file that will not end up in the built application, in addition to the `Importer-Info.plist` file you already noted. The two code files are located in the Importer subgroup of the Classes group in the Xcode project window.

The remaining items in the Finder project window are three code files, `main.m`, `MyDocument.h`, and `MyDocument.m`, and a special file named `Vermont_Recipes_Prefix.pch`. You'll find them in the Xcode project window in a moment.

5. Before closing the Finder project window, notice the `Vermont Recipes.xcodeproj` file. This is the file you double-click to open the project in Xcode any time you begin a new work session. It can be convenient to place an alias file pointing to it on the desktop or in your Favorites folder for easy access, or you can use Xcode's `File > Open Recent Project` menu item.

6. Close the Finder project window now. You will rarely need to open it again.
7. Continue exploring the Xcode project window. The Classes group is where you spend most of your time. Expand it now. You see the header file and implementation file, `MyDocument.h` and `MyDocument.m`, that you noted earlier. Select `MyDocument.h`, and its contents appear in the editing pane in the lower right of the project window. Double-click it, and it opens in a separate editing window (**Figure 1.4**).

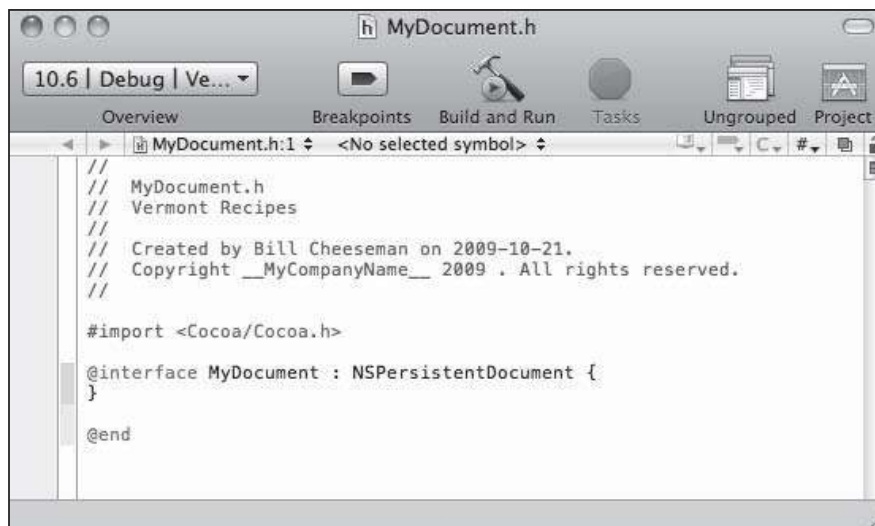


FIGURE 1.4
`MyDocument.h`
in a separate
editing window.

You also see an Importers subgroup, which you can expand to see some of the files in the Importers subfolder of the Finder project window. You can nest groups in the Xcode project window to keep items organized, without affecting their locations in the Finder.

8. You will spend much less time in the other groups, but explore them now to get an idea of their contents. The Other Sources group includes `main.m`, which is a small file that gets your application running. You won't alter it for Vermont Recipes, but there are circumstances in which it is useful to edit it. The Linked Frameworks subgroup of the Frameworks group holds references to all of the frameworks to which your product is linked. The Other Frameworks subgroup is simply a convenience, letting you use Xcode's search features to read system header files that are relevant to your project. Header files sometimes contain comments providing information about their usage that you can't find anywhere else, so it is helpful to have references to important Cocoa framework headers here, freeing you from having to dig for them deep in the System folder.
9. The Targets group contains an item for every target your project builds. Many projects contain only a single target, but more complex projects can contain a large number of targets, such as the Vermont Recipes Spotlight Importer you

see here. Expand the targets to see the several build phases for each of them. You will learn more about build phases later (**Figure 1.5**).

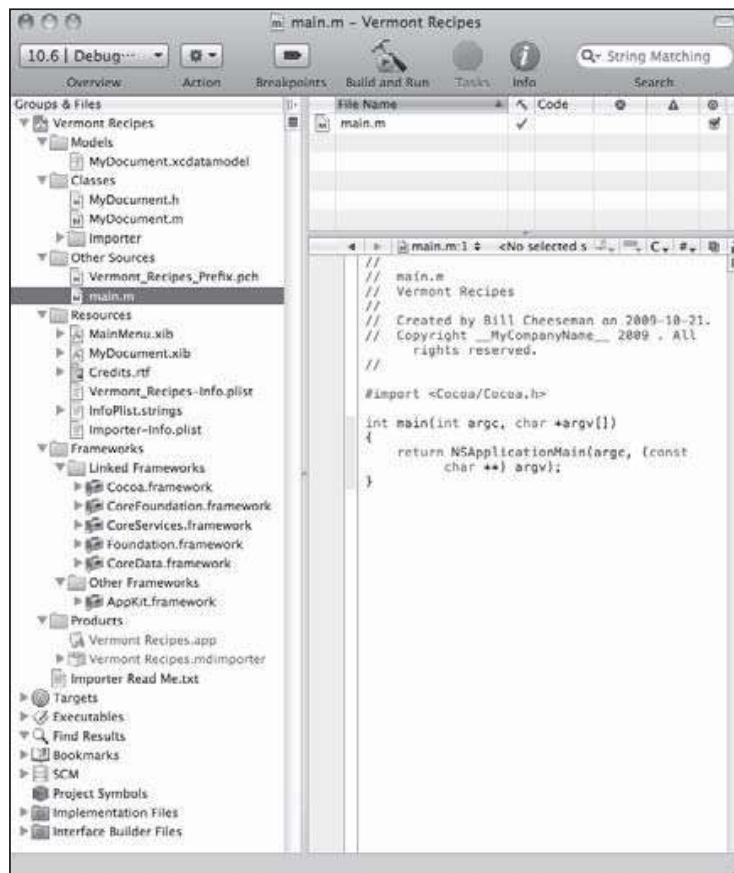


FIGURE 1.5
The Groups & Files pane fully expanded.

Step 3: Set Xcode Preferences

Take a quick tour of Xcode’s preferences. You don’t really have to change any of them for now, but it’s important to understand how Xcode is set up out of the box and the options it gives you.

Choose Preferences from the Xcode application menu. A familiar-looking preferences window opens with 13 buttons in the toolbar.

You configure your overall workspace in the General pane. For several years, I preferred to turn off the “Open counterparts in same editor” setting because I found it convenient to place header and implementation files for any given class side by side to make sure that I implemented the instance variables and methods I declared. I no longer do. While you’re getting started with Objective-C and the Cocoa frameworks, you might find it helpful to keep header and implementation files open side by side. You might also find it useful to select “Reopen projects on Xcode launch” and “Restore state of auxiliary windows” because you are likely to work only on this one project for a while.

The Code Sense pane controls Xcode's indexing and its ability to anticipate the names of common methods, autocompleting them as you type. You should keep the indexing setting enabled for all projects, to facilitate searching. You may also find it useful to show declarations in the Editor Function pop-up menu, which makes it easy to choose a specific method from a long list of methods declared in a header or implementation file, as well as method definitions. I have my own way of organizing my files, and I remember methods as much by location as by name, so I leave the alphabetical ordering option turned off. If you find code completion intrusive and distracting, turn it off by choosing Never in both pop-up menus, but more often than not it will save you a lot of time once you get used to it.

Use the Building pane to set up a common location where all of the project's built files are placed. By default, each project has its own build folder located in the project folder. I prefer to have a single, shared build folder for all of my projects located at ~/Documents/Projects/Cocoa. A lot of my work involves related projects with shared dependencies, and putting all the build products in a central location makes this much easier. It also simplifies archiving my work in progress, because I don't have to bother removing a build folder from the project to save space every time I compress and archive my code. Getting started, however, you will find it less confusing if you keep the build folder in the Vermont Recipes project folder. In the Build Results Window section, I normally set "Open during builds" to Always so that I can watch the build process unfold (**Figure 1.6**).

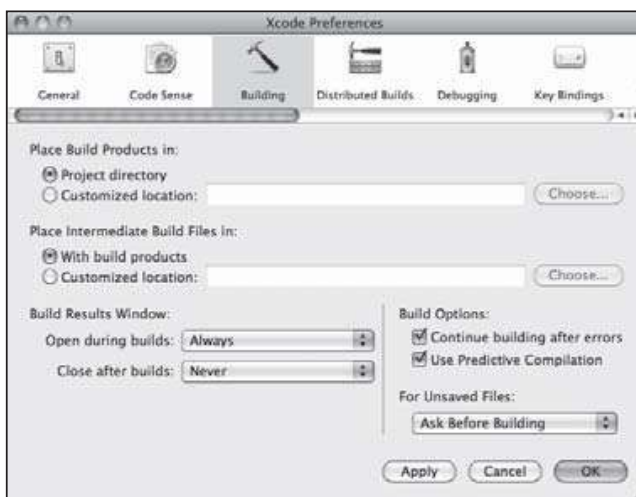


FIGURE 1.6 The Building pane in Xcode Preferences.

In the Indentation pane, I always turn on Line Wrapping by selecting the "Wrap lines in editor" checkbox, because I don't want to scroll horizontally to read a long statement in code. If you change this setting, click the Apply button; it takes effect immediately. Now the "Indent wrapped lines by" setting is enabled, and it is selected and set to 4 spaces by default. This is a problem, in my view, because the default tab and indent width are also set to 4 spaces. As a result, wrapped lines are indistinguishable from new lines visually. To make it obvious that a line is a continuation of a multiline wrapped statement, change the "Indent wrapped lines by" setting to 6 spaces. Now you can easily

detect a multiline wrapped statement because all lines after the first are a little more indented than lines that are indented in a code block (**Figure 1.7**).

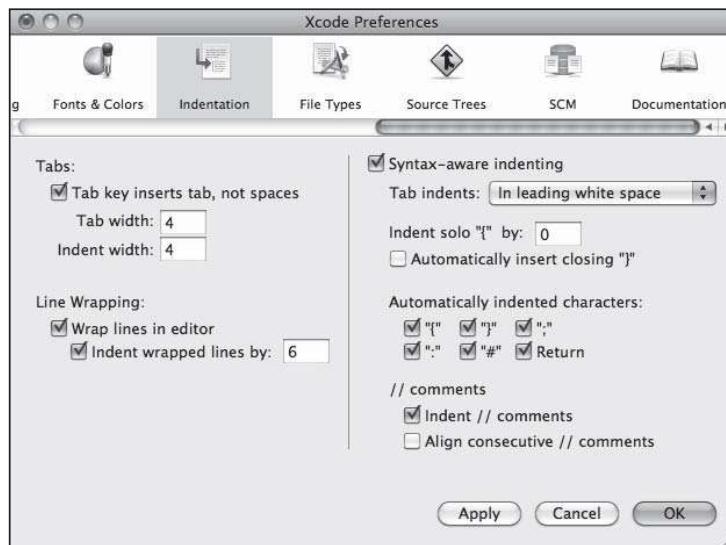


FIGURE 1.7 The Indentation pane in Xcode Preferences.

In the Documentation pane, if you are usually connected to the Internet with a broadband connection, it is important to keep the “Check for and install updates automatically” checkbox selected. Currently, Apple updates documentation periodically without announcing it. With this setting turned on, you automatically receive updated documentation in the background, if updates are available.

Step 4: Revise the Document’s Header and Implementation Files

Xcode templates typically generate a few files for you, to help you get started with the new project. Some of these files contain information identifying the file, the project, the developer, and the date, as well as providing the copyright notice required to protect your intellectual property. Xcode ferrets out most of the relevant information for you by scanning your computer, but you may nevertheless want to edit some of it.

When you create a project using a built-in template, the template often includes header and implementation files for a subclass that it derives from a standard Cocoa class. The document-based application template that you’re using, for example, creates a subclass of `NSPersistentDocument`, which in turn is a subclass of `NSDocument`. Your new subclass of `NSPersistentDocument` is declared in the `MyDocument.h` header file and implemented in the `MyDocument.m` implementation file, both of which are available in your new project window. You edit their initial contents in this step.

The NSDocument Class

Every document-based Cocoa application must subclass `NSDocument`. The document subclass controls the application's data model. It is considered a controller class in the MVC design pattern. You must subclass `NSDocument` to provide access to data in model objects or other repositories that hold your document's data, to provide accessor methods or properties that enable other objects to get and set the data values, and to provide methods that tell a window controller when the data has changed so that the user interface can be updated. Your document subclass usually does all this by controlling other model objects that you create and link to the document. The document is also the primary entry point for AppleScript.

Cocoa also has an `NSDocumentController` class, but it is not necessary to subclass it unless you need more than its default features. `NSDocumentController` manages a collection of documents and document types. When customization of `NSDocumentController`'s behavior is required, it is sometimes possible to use an application delegate instead of subclassing it. In *Vermont Recipes*, you will subclass it to add and manage an additional document type.

It may appear to you that a document-based application has two controllers in terms of the MVC design pattern. This is in fact the case. The subclass of `NSDocument` acts as a controller with respect to the application's model, where the data is located. At the same time, `NSWindowController` or a subclass of it acts as a controller with respect to the application's views, where the data is displayed and edited. This is not unusual. You will learn in this recipe and Recipe 2 that your subclasses of `NSDocument` and `NSWindowController` know how to talk to one another. They work together as a single controller object, in a sense. They are separated into two classes for several reasons. For example, a document needs only one subclass of `NSDocument` to define how it controls its data, but since the document may be associated with several different kinds of windows, it needs several different subclasses of `NSWindowController`, one for each kind of window, to define how it controls the windows. Multiple `NSWindowController` objects may also be useful even in an application that does not have different kinds of windows.

Read Apple's *NSDocument Class Reference* document for details about the methods it declares.

1. Expand the Classes group, and click the `MyDocument.h` header file to select it. You see its name and information about it in the upper-right pane of the Xcode project window, and its contents in the editing pane on the lower right. You can edit the contents of the file right in this window. Enlarge the project window, drag its vertical divider to the left, and move its horizontal divider up to make enough room to edit a large file.

My personal preference, however, is to edit files in separate windows, because I have two large monitors giving me room to compare and edit multiple files at once, each in its own window. An easy way to open a file in a separate window is to double-click the file in the Groups & Files pane, or Control-click (or right-click) it and choose “Open in Separate Editor” from the contextual menu. Another way to open a new window is to open the File menu, hold down the Option key to see its alternate menu items, and choose “Open in Separate Editor.”

2. Edit the information at the top of the `MyDocument.h` header file if it isn't what you want. Xcode is pretty good at pulling the information from various places on your system, including your company name for the copyright notice.

The template must provide a name for the new class, but it can provide only a generic name because it has no idea what you're up to. It names your new document class *MyDocument*, and it names the header and implementation files the same. This isn't very imaginative. It also isn't very descriptive, since your application will eventually create more than one kind of document. Change the name of the `MyDocument` class to `RecipesDocument` in this step. Changing the class name will lead to a number of ramifications covered in later steps.

Using standard Macintosh editing techniques, select **MyDocument** in the first full line of text, and change it to **RecipesDocument** so that the full name becomes *RecipesDocument.h*. In Step 5, you will change the name of the file itself in the Groups & Files pane to match, along with the names of several other files.

Next, add **2.0.0** to the end of the application's name, Vermont Recipes, in the second line so that it becomes *Vermont Recipes 2.0.0*. This is in fact version 2.0.0 of the Vermont Recipes application. If your application proves to have a long shelf life, it will likely go through many versions over a period of years, such as 2.0.1 and 3.9.9. You may find it convenient to have the version number at the top of the file when you have multiple versions open onscreen at once.

Change the third line if you aren't happy with Xcode's choice of your long user name as the developer.

Change the copyright notice in the fourth line. Putting on my attorney-at-law hat for a moment and assuming a professional pose, I advise you to enter my name, **Bill Cheeseman**, as the copyright holder. I also like to insert the international copyright symbol—the lowercase character *c* in a circle—after the word *Copyright*. Finally, change the date of the copyright to **2000-2009**. I wrote much of the code in Vermont Recipes 1 during the period from 2000 to 2002, and I added most of the new code for Vermont Recipes 2 in 2009.

The only code that requires editing now is the name following the `@interface` directive. Change it from `MyDocument` to `RecipesDocument`. The `@interface` directive

declares the name of the class. While it is possible to declare multiple classes in one file and to give the file an altogether unrelated name, the more common practice is to declare one class per file and to give the class and the file the same name.

3. Make corresponding changes to the information at the top of the implementation file. Open the `MyDocument.m` implementation file in a separate window so that you can see its contents and the contents of the header file at the same time. Edit the information at the top of the implementation file to correspond to the changes you made to the header file. Leave the `.m` file extension as it is, to distinguish this implementation file from the header file with its `.h` extension. Be sure to change the `#import` preprocessor directive to `#import "RecipesDocument.h"` (with the quotation marks), so that the preprocessor can find the header file when you build the project. Also change the `@implementation` directive to `RecipesDocument`, to match the class name in the `@interface` directive in the header file.
4. Take a look at the rest of the implementation file. It contains the code or code stubs for three Objective-C methods, `-init`, `-windowNibName` and `-windowControllerDidLoadNib:`. The template that Xcode selected when you chose the “Create document-based application” setting provided this code. The template does not declare these methods in the header file because they are already declared in the Cocoa framework headers. Many methods declared in the Cocoa frameworks are not implemented in Cocoa or are implemented there only in a default fashion, and you are expected to provide a custom implementation. In such circumstances, most developers don’t re-declare the method in their own header files, although there is nothing to stop you from doing so if you prefer.

One of the methods, `-windowNibName`, returns the `NSString` object `@“MyDocument”`. If you leave this statement as is, the application will fail to load the document window’s nib file when you launch it, because in a moment you are going to change the name of the nib file. Change the `return @"MyDocument"` statement to `return @"RecipesWindow"` (with the quotation marks). You will delete this method in Step 6, but for now you should make this change for the sake of consistency.

5. To be completely consistent, change the information at the top of the third code file that the document-based application template created, `main.m`, as well. To find it in the Groups & Files pane, expand the Other Sources group.
6. Close the `MyDocument` header and implementation files, and `main.m`, so that they are out of the way while you change the filenames of the first two. You will return to them for more editing in Step 6.

Step 5: Rename the Document's Files

First, follow up on the changes you made to the contents of the header and implementation files by changing their filenames. Nothing requires header and implementation filenames to correspond to the name of the class they represent, but it is usually easier to understand a complicated project if the names do match. Changing filenames in Xcode projects is a delicate task. You must always take care to find every place in the project where the name needs to be changed, and you must get the spelling right.

This task is somewhat tedious, but it would be even more tedious if you left it until later when more changes might have to be made. If it were only a matter of changing the header and implementation filenames, this would be easy. However, the document-based application template with Core Data support creates a number of other files that bear the same name. The convenience of having them created for you automatically is offset by the need to change their names manually.

Changing filenames in Xcode projects tends to follow this pattern. You couldn't avoid the job in this case because the template supplied the original name without asking you. The moral for the future, however, is to think carefully beforehand about the names you give your files, because changing them is a painful and error-prone process.

1. You can't simply double-click the filename in the Groups & Files pane to select its text for editing, as you normally do when editing text within a file, because that gesture causes Xcode to open the file itself to let you edit its contents. Instead, as in the Finder, click the filename once, pause long enough that Xcode doesn't interpret the next click as a double-click, and then click again. Alternatively, Control-click the `MyDocument.h` header file in the Groups & Files pane and choose **Rename** from the contextual menu. When the name of the file is ready for editing, change it to **RecipesDocument.h**, and then press Enter or click elsewhere in the window to commit the change. To verify that the file's name was really changed, open the Finder project window and see that, sure enough, the header file is now named `RecipesDocument.h`.
2. There are several other files remaining to rename. Normally, you change project filenames in the Xcode project window, not the Finder project window, as you did just now with the header file.

I nevertheless find myself accidentally using the Finder to change filenames from time to time, and you will probably do that, too. It's easy to recover from this harmless mistake. To see how, use the Finder now to change the name of the `MyDocument.m` implementation file to **RecipesDocument.m**. When

you're done, click in the Xcode project window and see that the name of the `MyDocument.m` file in the Groups & Files pane did not automatically change to match the new Finder name. Instead, it turned orange, signaling that Xcode can no longer find a file named `MyDocument.m`.

To point Xcode to the renamed file, use the contextual menu on the red file name and choose `Get Info`, and then click the `Choose` button to locate it. If you moved the file in the Finder instead of renaming it in place, choose `Project > Add to Project`, navigate to its new location in the `Open` panel, select it, and click `Add`. Files already in the project are dimmed, but the misplaced file is selectable.

If you use the `Add to Project` menu item, after you click `Add`, the `Open` panel closes and a new sheet appears with a number of settings you can reconfigure before adding the file to the project. It is a good idea to leave its “Copy items into destination group’s folder (if needed)” checkbox selected. However, if the file is already in the correct folder but was renamed, you can leave it deselected. Unless you’ve been fiddling with this sheet previously, its other settings should be correct. Click `Add`, and see that the `RecipesDocument.m` implementation file is now included in the `Classes` group in the Groups & Files pane.

Delete the orange reference to `MyDocument.m` by selecting it and pressing the `Delete` key.

3. Use the Xcode project window to rename the remaining `MyDocument` files. In the `Models` group, change `MyDocument.xcdatamodel` to **`RecipesDocument.xcdatamodel`**. In the `Resources` group, change `MyDocument.xib` to **`RecipesWindow.xib`**.
4. There is one more place where the name must be changed in Xcode. (You’ll find yet another in Interface Builder, in Recipe 2.) You won’t find it by scanning filenames. In the `Resources` group, open `Vermont_Recipes-Info.plist`. This is a property list file, a text-based file in XML format, and it opens in a built-in Xcode property list editing window. Be careful where you click in this window, because it’s easy to change settings without realizing it.

Click the “Document types” disclosure triangle to expand it; then expand each of the subsidiary entries, `Item 0 (Binary)`, `Item 1 (SQLite)`, and `Item 2 (XML)`. In each, find the `Cocoa NSDocument Class` key on the left, double-click the corresponding value entry on the right to edit it, and change the value from `MyDocument` to **`RecipesDocument`**. When you’re done, close the `Vermont_Recipes-Info.plist` file. You will revisit it in Step 9.

Step 6: Edit the Document's Methods

Now reopen the newly renamed *RecipesDocument.m* implementation file to make some more changes to the method implementations provided by the template.

The Vermont Recipes 2 application specification calls for a complex Core Data document-based application with multiple documents and windows. You learned at the beginning of this recipe that the document-based application template you selected when you created the project assumes you are writing a single-document, single-window application. The time has now come to make the changes to the template-supplied methods that Apple recommends for multidocument, multi-window applications.

The *MyDocument.m* implementation file provided by the document-based application template, which you have just renamed *RecipesDocument.m*, originally contained three methods that override methods built into the *NSPersistentDocument* class or its superclasses. They are invoked automatically at appropriate times by the Cocoa frameworks. Your overrides of them are passive, in a manner of speaking. You rarely or never call these override methods yourself; instead, you modify them so that when Cocoa calls them, they will perform custom tasks or provide custom information that is unique to your application. For example, the `-windowNibName` method should return the name you gave to the nib file for the document's main window.

These three methods are the minimum that must be overridden to build a working Cocoa document-based application. In fact, if you were willing to live with the minimal capabilities of these methods, you could build and run the Vermont Recipes application without changing the names of the document and nib files supplied by the template and without writing any code of your own. You would discover that it actually works, up to a point. For example, if you launched the application, it would automatically open the main document window. If you chose `File > New`, it would open another, identical document window.

This would be a short book if you were satisfied with the template as is. In fact, you'll make many changes to it in Vermont Recipes. In this step, you delete two of the supplied methods, `-windowNibName` and `-windowControllerDidLoadNib:`, both of which have to do with the way the application loads and uses the nib file. This is the first real Cocoa code you will write in this book.

The first method overrides *NSDocument*'s `-windowNibName` method. Its only purpose is to return the name of the document's nib file (without the *nib* extension), which you supplied by editing your override of the method. In the template, the name supplied was `@MyDocument`. You changed it to `@RecipesWindow` in Step 5 for the sake of consistency, as if you planned to use this method in the Vermont Recipes

application. `NSPersistentDocument`'s superclass, `NSDocument`, uses the name that `-windowNibName` returns to instantiate a window controller for the document. In a simple document-based application, you don't write a custom version of the window controller; instead, the `NSWindowController` class is used as is.

However, because the Vermont Recipes application specification calls for a multi-document, multiwindow application, you need to delete the `-windowNibName` method here and override `NSDocument`'s `-makeWindowControllers` method instead. In `-makeWindowControllers`, you instantiate one or more custom window controllers that you design to handle the more complex features of the Vermont Recipes application. Each window controller created in `-makeWindowControllers` is added to a list maintained by the document using the `-addWindowController:` method.

You delete the `-windowControllerDidLoadNib:` method supplied by the template as well. This is a delegate method that Cocoa calls, if you implement it, immediately after a nib file loads—but only if the document owns the nib file. It is often used in simple document-based applications, but it won't work in Vermont Recipes because the new window controller will own the nib file.

The changes described in this step are detailed in Apple's *Document-Based Applications Overview* document. This approach conforms to established Cocoa tradition. Apple does not provide a template for a more complex document-based application like Vermont Recipes.

1. Open the `RecipesDocument.m` implementation file. Immediately below the line reading `#import "RecipesDocument.h"`, add this line:

```
#import "RecipesWindowController.h"
```

Without this, the override of `-makeWindowControllers` in instruction 3, below, would generate a compiler warning because the compiler wouldn't see a declaration for that method. You will create the `RecipesWindowController` header and implementation files in the next step.

2. Delete the `-windowNibName` and `-windowControllerDidLoadNib:` methods in their entirety.
3. Replace the deleted methods with this new `-makeWindowControllers` method:

```
- (void)makeWindowControllers {
    RecipesWindowController *controller =
        [[RecipesWindowController alloc] initWithWindowNibName:
            @"RecipesWindow"];
    [self addWindowController:controller];
    [controller release];
}
```

Cocoa invokes this method automatically every time the user opens a data-base document—for example, by launching the application or choosing File > New. The first statement in the body of the method is in a form that you will encounter very frequently in your work. It combines in a single statement the allocation of memory for the new window controller and the initialization of that object. The second statement inserts the new window controller into a built-in array of window controllers maintained by every `NSDocument` object, providing access to all open windows associated with the document. The third statement releases the object that was just created; you no longer need it here because the `-addWindowController:` method retained it when adding it to the window controllers array. As you will read later, the correct way to think of this is that you assumed ownership of the object when you instantiated it, so you must release it when you're done with it. When it was placed in the array, the array assumed ownership of it, and the array will assume responsibility for memory management hereafter.

An important feature of the `-makeWindowControllers` method is that it initializes the new controller by passing the name of the nib file (without the file extension) into the initializer in the first statement. The `-windowNibName` method provided by the template originally filled the role of telling Cocoa what nib file to load, but you've just deleted that method. The new `-makeWindowControllers` method now fills the same role, providing the nib file's name. The difference is that you can modify `-makeWindowControllers` later so that the document opens additional windows and their nib files, simply by adding additional statements that allocate additional window controllers and initialize them with different nib filenames.

Looking ahead for a moment, you will learn in Recipe 3 that it is usually better to let the window controller specify the name of its associated nib file. To prepare for that way of doing things, you would initialize the window controller here by calling its `-init` method instead of its `-initWithWindowNibName:` method, and you would later implement the `-init` method in the window controller so that it calls `-initWithWindowNibName:`. For now, leave the `-makeWindowControllers` method as is for the sake of keeping things simple.

If you look up the description of the `-addWindowController:` method in Apple's *NSDocument Class Reference* document, you'll see a reference to `NSWindowController`'s `-setDocument:` method. The description of `-setDocument:` reveals that `-addWindowController:` calls it automatically to set up the window controller's link back to the document. Because of these mutual links between the document and the window controller, they are able to send messages to one another. The window controller can send messages to the document when the

user performs some action in the application's user interface, and if some change occurs in the document (for example, through an AppleScript command), the document can send messages through its window controllers array to tell the window controller to update the user interface. This communication back and forth between the document and the window controller can be thought of as half of the MVC design pattern, since the document is a gateway to the MVC model. You will implement the other half—communication between the window controller and the window, the MVC view—in Recipe 2 using Interface Builder.

In this one simple `-makeWindowControllers` method, you have encountered a host of other Objective-C and Cocoa techniques that require explanation. In this brief introduction, however, take it on faith that it will work. Once you've got the beginnings of the application up and running, you will pause in Recipe 3 to review what you've accomplished.

4. Close the `RecipesDocument.m` implementation file to get it out of the way.

Step 7: Create and Revise the Window Controller Files

In a very simple single-document application, `NSWindowController` can perform its job behind the scenes without subclassing. However, in a multidocument, multiwindow application, you typically need more power from the window controller. The `RecipesDocument` class would become inordinately complicated if it were directly responsible for controlling all of the windows and their contents. It is easier to organize and maintain the application if each window has its own window controller.

In a complex document-based application like Vermont Recipes, you instantiate each document window in its own separate nib file, and you create a unique custom window controller subclass to act as the file's owner of each nib file. In this step, you create `RecipesWindowController`, a subclass of `NSWindowController`, to handle the application's main database window. Later, in Recipe 2, you will revise the `RecipesWindow` nib file, which was created by the document-based application template under the name `MyDocument.xib`, to make this new `RecipesWindowController` subclass the nib file's owner.

The NSWindowController Class

In a document-based Cocoa application using the MVC design pattern, the controller classes are especially important.

In a sense, the controller is the least standardized of the MVC elements, because it must keep a unique set of models and views synchronized and tell them how to perform the unique functions of the application. A controller manages a window and its views on behalf of a document. A model class representing the application's data should know nothing about the GUI; and the view classes, such as windows, tab views, and controls, should know nothing about the specific data they represent. You don't have to subclass NSWindowController for a very simple application, but in a typical document-based application, you usually do.

Only the controller knows how the model and the views interact. The document—which is a controller in its own right, as you learned earlier—tells the window controller when data has changed (perhaps the user has reverted to the document's saved state or chosen Undo or Redo from the Edit menu, or an AppleScript script has altered some data), and the controller then tells the views to change their state to reflect the new data. Similarly, the views tell the window controller when the user has changed the state of a control (perhaps the user has selected or deselected a checkbox), and the window controller tells the document—another controller—to tell the model to update its data stores accordingly. For these reasons, you will typically write much more custom code for controller classes than for the model or the views.

Modern software engineering offers many reasons why an application's data and user interface should be factored out into separate classes in this fashion. It has mostly to do with keeping concepts clear and easing maintenance and modification.

But there are also specific reasons relating to the way Mac OS X works. In particular, the GUI is not the only interface of most Macintosh applications. There is also a scripting interface. Using AppleScript, a user can command an application to alter its data without touching a control. In doing so, AppleScript does not need to know anything about the GUI; it can communicate, and usually it should communicate, directly with the application's model, and it may do so without requiring that a window be open. By keeping the model separate from the views and relying on the controller to mediate between them, Cocoa's AppKit can give you AppleScript support almost for free. Just as the user's reverting a document to its saved state tells the window controller to update any affected views, so does a script's alteration of the document's data invoke the same methods, with the same effect on the views.

Another important feature of a window controller is its role as a delegate of other classes. In the Cocoa environment, many classes perform their work by

(continues on next page)

The NSWindowController Class *(continued)*

invoking *delegate methods*, which you as application designer can choose to implement in your subclasses. This is one of the abilities that allow Cocoa to provide so much precoded functionality while preserving the flexibility to let the application designer create unique applications. Your NSWindowController subclass is one class that plays an important role as a delegate in the Cocoa scheme of things, as you will see.

Read Apple's *NSWindowController Class Reference* document for details about the methods it declares.

In changing the nib file's owner, you will discover that RecipesDocument no longer needs a nib file at all, because it no longer has anything to do with how its data is displayed. This approach—separating the document from the user interface and inserting a window controller to handle all communications between the document and the window—is squarely in line with the MVC design pattern. The document and window controller subclasses act in tandem as a controller mediating between the application's data and its views. The document subclass focuses on the data, while the window subclasses focus on the views, and the document and the window subclasses communicate with one another throughout the process in order to keep everything synchronized.

1. In Xcode, choose File > New File to select a template.
2. Explore the New File window. Like the New Project window you encountered in Step 1, which it closely resembles, it has a left pane where you choose basic categories of files, and panes on the right where you choose a specific template and configure it.
3. Click Cocoa Class in the left pane; then click the “Objective-C class” template's icon in the upper-right pane. In the lower-right pane, use the “Subclass of” pop-up menu to create a subclass of NSWindowController (**Figure 1.8**).

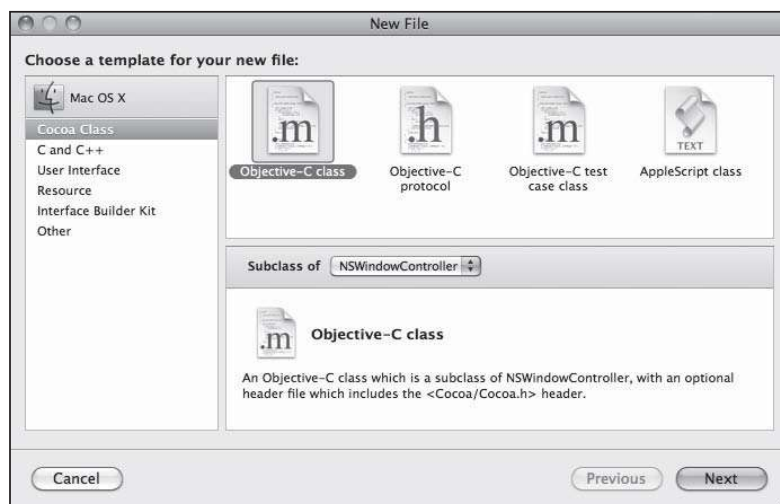


FIGURE 1.8
Setting up Xcode's
New File window.

4. Click the Next button. In the next window in the New File assistant, fill in the File Name field for the implementation file by typing **RecipesWindowController**, so that it reads `RecipesWindowController.m`. Leave the checkbox below the File Name field selected so that you will also create a header file of the same name with the `.h` file extension (**Figure 1.9**).

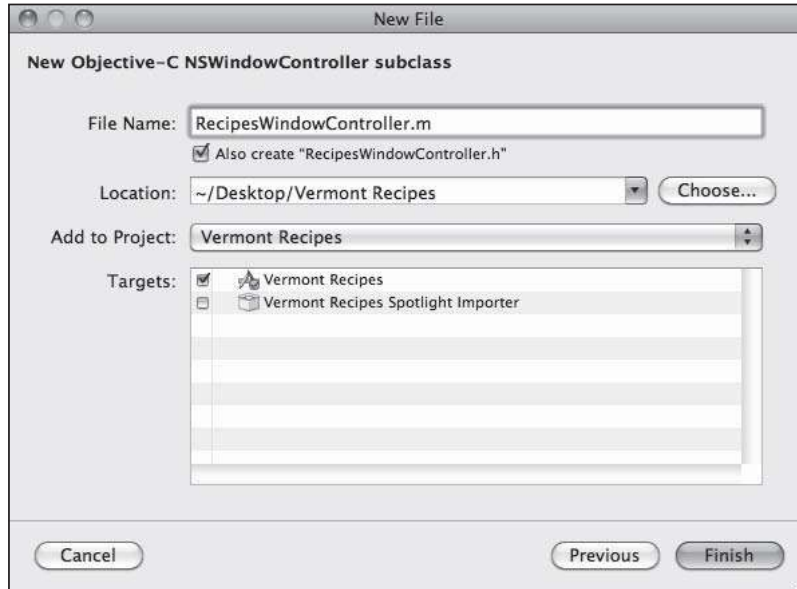


FIGURE 1.9
Finishing Xcode's
New File window.

5. Click the Finish button to create the new files in the Vermont Recipes project and its Vermont Recipes target. In the Xcode project window, you see the two new files. If they aren't located in the Classes group in the Groups & Files pane, drag them there.
6. Open the new header and implementation files and examine them. You see that the `@interface` and `@implementation` directives have been set up with a class name of `RecipesWindowController` to match the filename.
7. Edit the identifying information at the top of the header and implementation files following the patterns you established in Step 4. You may only have to add **2.0.0** to the reference to Vermont Recipes.
8. The template did not insert any methods. Leave the files empty for the time being.
9. Close the `RecipesWindowController` header and implementation file windows. You don't have to save them at this point, although Xcode might ask you to save them. Unsaved files appear darker than normal in the Groups & Files pane as a constant reminder. You will be reminded to save them again later, when you build the project or close the main project window. If you are the cautious type, you can choose `File > Save` before closing each of them.

Step 8: Edit the Credits File

The document-based application template created several other files in the project that require editing. Start with the Credits.rtf file, which contains the information the user sees when choosing About Vermont Recipes from the application menu.

1. Expand the Resources group in the Groups & Files pane of the Xcode project window, and click or double-click the Credits.rtf file to open it for editing.

You see that it contains several headings along with humorous placeholders where you can identify people who contributed to the application.

Many developers delete this text and enter in its place a short description of the application. For an example, choose About Xcode from the Xcode application menu. The About window that opens contains Xcode's icon, name, version, and copyright obtained from other sources within the Xcode application package, as well as information about the versions of the Xcode components. Xcode obtained the component information from its Credits file.

Since I am responsible for Vermont Recipes, it should come as no surprise that I credit myself for everything. I also give thanks to the Stepwise team because of their important contributions to the success of the original Vermont Recipes venture and to Michael Tsai for his extraordinary work as technical editor of the second edition.

Enter the following text in Credits.rtf, replacing the file's default contents. Because it is a Rich Text Format (RTF) file, you can do some minimal formatting, such as making the headings boldface.

Vermont Recipes 2 is based on Cocoa Recipes for Mac OS X—The Vermont Recipes (Second Edition, Peachpit Press, 2010). For more information, visit www.peachpit.com.

Engineering:

Bill Cheeseman
Quechee Software
www.quecheesoftware.com

Human Interface Design:

Bill Cheeseman

Testing:

Bill Cheeseman
Michael Tsai

(continues on next page)

Documentation:

Bill Cheeseman

With special thanks to:

The Stepwise team

Michael Tsai

This text is too long to fit in a small About window. When you build and run the application at the end of this recipe, you will discover that Cocoa automatically installs it in a scrolling text view so that the user can read all of it.

2. You don't have to use RTF, as the Credits.rtf file provided by the template does. Cocoa also supports use of an HTML file for more elaborate formatting. You can use links in an RTF Credits file using the contextual menu, and any HTML links you insert in a separate Credits.html file will also become clickable links in the application's About window. If you have a Web site for your product, you should certainly include a link in the About window.

Credits.html will take precedence over Credits.rtf if both are present. Leaving the RTF version in the file is necessary in the unlikely event you jump through the hoops required to write an application that can run under Mac OS X 10.0 Cheetah, because HTML Credits files were not supported in Cheetah. I habitually leave the RTF version in my applications for old time's sake.

Create a new Credits.html file using the same techniques you used in Step 7 to create the new window controller files. Instead of selecting the "Objective-C class" template in the Cocoa Class pane of the initial New File window, as you did in Step 7, select the Other category and the Empty File template. It is an empty text file. Click Next, enter **Credits.html** in the File Name field, click the Choose button and change the file's location to the project's English.lproj folder, and click Finish. In the Xcode project window, you may have to drag the new Credits.html file into the Resources group and drop it next to Credits.rtf. Click or double-click it to open it for editing. Copy and paste the contents of your modified Credits.rtf file into the new HTML file, and you're almost done.

You can use your favorite Web authoring tool to create a fancy About window, but in Vermont Recipes you'll be content to provide simple formatting and to add clickable links to this book's publisher, Peachpit Press, and to my company, Quechee Software. Insert HTML formatting tags as shown here, and then save the file.

```
<p>Vermont Recipes 2 is based on <i>Cocoa Recipes for Mac OS  
X&#151;The Vermont Recipes</i> (Second Edition, Peachpit  
Press, 2010). For more information, visit <a  
href="http://www.peachpit.com">www.peachpit.com</a>.</p>
```

```
<p><b>Engineering:</b><br>
Bill Cheeseman<br>
Quechee Software<br>
<a href="http://www.quecheesoftware.com">
    www.quecheesoftware.com</a></p>
<p><b>Human Interface Design:</b><br>
Bill Cheeseman</p>
<p><b>Testing:</b><br>
Bill Cheeseman<br>
Michael Tsai</p>
<p><b>Documentation:</b><br>
Bill Cheeseman</p>
<p><b>With special thanks to:</b><br>
The Stepwise team<br>
Michael Tsai</p>
```

If you were to build and run the application now, the About window would work as expected. You would see two problems, however: Near the top, the window would claim that this is version 1.0 of Vermont Recipes, and the copyright notice you expect to see at the bottom would be missing. You'll fix these problems in Step 9.

Step 9: Edit the Info.plist File

When you explored the Xcode project window in Step 2, one of the files you saw in the Resources group in the Groups & Files pane was `Vermont_Recipes-Info.plist`. You made one change to its contents at the end of Step 5. In this step, you examine the remainder of the file's contents and edit some more entries. This is a tedious and technical process, but you have to do it in every new project.

Xcode saves the `Vermont_Recipes-Info.plist` file in the built application package under the generic name *Info.plist*. For this reason, it is customary to refer to it simply as the Info.plist file. When building a simple application from scratch, you may use the generic name in the project folder. However, the document-based application template gives it the more specific name to distinguish it from a second Info.plist file in this project, `Importer-Info.plist`. One of the fields in the project's build settings tells Xcode which of these files it should save as the application's main Info.plist file, as you will see in Step 12.

The Info.plist file is in XML property list format, containing key-value pairs in human-readable plain text. It should be saved in UTF-8 encoding. You normally edit the entries in Xcode's built-in property list editor. You can edit the file in any text editor as well, where it appears in the form of plain text with XML tags, but this requires more work on your part to maintain the correct syntax. Apple defines several different property list formats for use when developing applications.

Cocoa uses the Info.plist file for a variety of purposes, including to provide the short application name that appears in the menu bar when the application is running; to tell the system where the application and document icons are located; to provide the version, copyright, and other strings used in the Finder's Get Info window, the About window, and other dialogs and alerts; and to specify document types the Finder uses to open the application when the user double-clicks one of its document icons. Xcode also places the type and creator code for your application into a file named PkgInfo in the compiled application bundle.

1. Open the Vermont_Recipes-Info.plist file for editing in Xcode. In the Info.plist window, you see over a dozen entries that were supplied by the document-based application template. Some of them are appropriate as is, and you won't change them. Others are placeholders and require customization for the Vermont Recipes application (**Figure 1.10**).

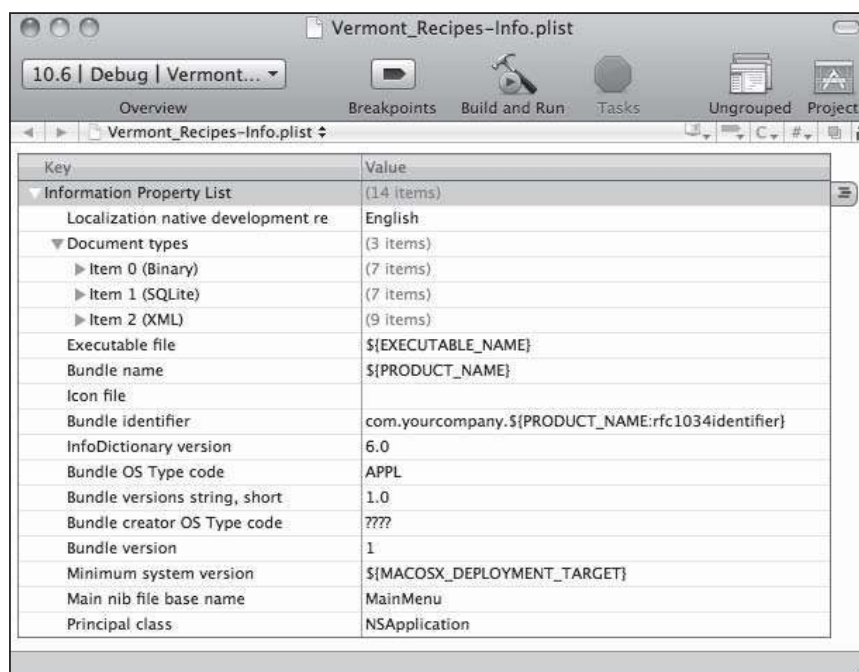


FIGURE 1.10
Info.plist open in Xcode's property list editor.

2. Choose View > Property List Type in the Xcode menu bar. You see four types of property list files in a submenu, and the Info.plist format is currently set. This is correct, so leave it alone.
3. Control-click anywhere in the Info.plist window to open a contextual menu. Choose Show Raw Keys/Values, and all of the keys in the left column change to

the technical names you use in code. For example, the “Document types” key becomes `CFBundleDocumentTypes`, a Core Foundation setting, and “Main nib file base name” becomes `NSMainNibFile`, a Cocoa setting. I recommend that you always work with the raw key names. This is the terminology you should use when you look them up in documentation and when you write code.

4. Click the disclosure triangle to expand the `CFBundleDocumentTypes` key in the left column, and then expand each of the Binary, XML, and SQLite document types. If you didn’t already explore them in Step 5 when you renamed the `NSDocumentClass` item, explore one of them now, including its expandable subsidiary entries. These entries control important features of the documents in which the application will save its data. You will learn more about them later.

For a released Core Data application with large and complex storage needs, Apple recommends that you use the SQLite type for maximum efficiency. The Binary type is for documents with more modest storage requirements where access speed is the paramount consideration. For development, you should use the XML type because it is human readable and you can check your work.

In Vermont Recipes, you will use the SQLite document type, not the Binary type. Select the Binary type entry and press the Delete key.

Keep the XML document type for the time being to help in development. Drag the entire Item 1 (XML) entry upward so that it is first in the list under the `CFBundleDocumentTypes` key. It becomes *Item 0 (XML)*. Cocoa automatically opens the first type in the list unless told otherwise. Remember to delete the XML type before releasing Vermont Recipes 2 to the public, or leave it there and move it back to the bottom of the list if you decide to give users the option to save data in XML format.

Apple’s *Information Property List Key Reference* states that every document type should include at least one of the keys `LSItemContentTypes`, `CFBundleTypeExtensions`, `CFBundleTypeMIMETypes`, and `CFBundleTypeOSTypes`. The `Info.plist` file provided by the document-based application template meets this requirement because it includes several of these keys. However, *Information Property List Key Reference* also advises that `LSItemContentTypes` takes precedence over the others in Leopard and Snow Leopard, and it is the most modern key. For example, it plays an important role in QuickLook. The `LSItemContentTypes` key is not automatically included in the template, so you’ll have to add it manually. This takes several steps.

- a. Delete the `CFBundleTypeExtensions`, `CFBundleTypeMIMETypes`, and `CFBundleTypeOSTypes` entries, since they are ignored when `LSItemContentTypes` is present and, in fact, are deprecated as of Leopard.

- b. With the Item 0 (XML) document type selected and its subsidiary entries showing, click the outline button to the right of the Item 0 (XML) entry. A new subsidiary entry appears with an open combo box displaying a list of available keys. Choose `LSItemContentTypes`.
- c. Expand the `LSItemContentTypes` entry to show Item 0; then tab to its Values field and enter **public.xml** for now. This is a Uniform Type Identifier (UTI), which has become a key feature in the mechanism that Mac OS X uses to associate documents with the application that can open them.
- d. *Information Property List Key Reference* suggests that you should include the `NSExportableTypes` key whenever you use `LSItemContentTypes`. Therefore, select the `LSItemContentTypes` entry; click the Add (+) button beside it to add a new, blank entry; choose `NSExportableTypes` from the combo box; and enter **public.xml** as its value. (If `NSExportableTypes` does not appear in the combo box list, type it in the Key field yourself, use the contextual menu to change the entry type to Array, create a new Item 0, and enter **public.xml** as its value.)

The Guidelines are actually misleading. You do not need to export the content type when the UTI is a public type declared by Apple. You will come back to this entry later and change it to a proprietary UTI, which you must export. Leave it as `public.xml` for now.

- e. *Information Property List Key Reference* also suggests that you include the `LSHandlerRank` key with `LSItemContentTypes`. Therefore, select the `NSExportableTypes` entry; click the Add (+) button beside it to add a new, blank entry; choose `LSHandlerRank` from the combo box; and choose Alternate from the pop-up menu in the Values column. This means that Launch Services will not use Vermont Recipes 2 to open other XML files on your computer because Vermont Recipes 2 does not own the XML document type.
 - f. The `LSTypeIsPackage` entry should be deleted. The `CFBundleTypeName` entry should be set to “Vermont Recipes Database.” Leave the `CFBundleTypeIconFile` entry in place, but you won’t enter its value until later.
 - g. Repeat steps 4.a.–f. in the Item 1 (SQLite) document type entry, and for now use **public.database** for the value of Item 0 of its new `LSItemContentTypes` and `NSExportableTypes` entries.
5. You don’t have an application icon yet, so leave the `CFBundleIconFile` entry blank.
 6. Move along to the `CFBundleIdentifier` key. An application’s identifier, in the form of a reverse-DNS string, has become another key feature in the mechanism that Mac OS X uses to associate documents with the application that owns

them, largely supplanting and, in Snow Leopard, replacing the old creator type mechanism from Mac OS 9 and earlier.

The identifier supplied by the template contains a placeholder. To edit the identifier, enter **com**, followed by a dot, followed by a company name or any other name you use to identify the source of the product, followed by another dot, and followed finally by a reference to the application name. In the default identifier supplied by the template, the product name reference is in the form of a variable expansion, `${PRODUCT_NAME:rfc1034identifier}`. The `PRODUCT_NAME` variable was automatically set to Vermont Recipes when you created the project. The reference to *rfc1034identifier* causes Xcode to replace the space with a hyphen to comply with requirements for UTIs.

This is a reverse Domain Name System (DNS) name that uniquely identifies your application within the Macintosh community. Apple does not have to maintain a registry to police the uniqueness of your identifier, because trade name, trademark, and other laws likely protect your company name, and the identifier incorporates a product name that you control.

I consider Vermont Recipes to be a Quechee Software product, and I have registered *Quechee Software* as a trade name with the Vermont Secretary of State. You should therefore enter the identifier in this form:

com.quecheesoftware.\${PRODUCT_NAME:rfc1034identifier}.

The identifier comes out in the built application as `com.quecheesoftware.Vermont-Recipes`.

7. The value of the `CFBundleShortVersionString` as supplied by the template is 1.0. You are now writing version 2 of Vermont Recipes, so change the value to **2.0.0**. This will cause the About window to show the application version as 2.0.0, instead of the incorrect 1.0 you saw in Step 8. It will also appear in the Finder's Get Info window and in other places.

The three digits separated by dots in the version number represent, from left to right, the major release version, the minor release version, and the maintenance release version. Most developers have traditionally shown the maintenance release version only when it is greater than 0, such as 2.0.1. However, current Apple documentation indicates that all three version numbers are “required” in all cases, even for a major release such as 2.0.0. See Apple's description of the `CFBundleShortVersionString` setting in *Information Property List Key Reference* and its “Document Revision History” section.

As the name of this key suggests, it is not a number but a string. You can therefore include more information in the value. Many developers include customary codes indicating that this is a development (*d*), alpha (*a*), beta (*b*),

or release candidate (*rc*) version. If you choose to follow this practice, 2.0.0d1 or 2.0.0a1 might be appropriate for Vermont Recipes, since the *beta* designation is typically reserved for applications whose feature list is frozen. However, Apple's description of this setting in the *Information Property List Key Reference* strongly suggests that it should be used only to identify a released iteration of the application. All versions of Vermont Recipes released with this book therefore use the simple three-digit form 2.0.0, distinguishing prerelease builds by relying on the `CFBundleVersion` entry described next.

`CFBundleShortVersionString` is called a *short* version string for a reason—namely, to allow it to fit within small spaces, such as the Version column in a Finder window shown in list view. Some applications incorrectly include the name of the application in the short version string. If you provide a short version string that is too long, the Finder will truncate it in the Version column, to the annoyance of your users who can't make out the version of the application.

8. Note the `CFBundleVersion` entry. This is set to 1 by the template. It appears in the application's About window in parentheses following the short version string, as in *2.0.0 (1)*. Although there are many ways to use this key, it is common to increment its value every time you build even a fairly modest change to the application. For example, in a large organization, you can use it to mark internal release versions. In Vermont Recipes, you will increment it once at the beginning of every recipe, so that the About window will show you at a glance which recipe is responsible for the version you are running at any given time.

You could continue to increment the bundle version even after you update the short version string for a new major version of the application. You would then be able to track every version of the application using a single variable, without having to test the short version string as well. In this recipe, however, you'll leave the Vermont Recipes 2 bundle version set to 1 for the convenience of knowing which recipe each build of the application represents.

9. The next entry to change is `CFBundleSignature`. This and the `CFBundlePackageType` entry hold the file and creator codes for the application. Each of these is a 4-byte value consisting of four standard C char constants, written as four characters representing an integer value. They are encoded in a small file in the Contents folder of the application package, `PkgInfo`, when you build the application.

The `CFBundlePackageType` entry should remain `APPL`, for *application*, but change the `CFBundleSignature` value to **VRa2**. This is an arbitrary value that I made up. *VR* stands for *Vermont Recipes*, of course, and in my thinking, *a* stands for *application*. The final digit is the major version number. The creator code for Vermont Recipes 1 was *VRa1*, and I registered it with Apple several years ago to ensure its uniqueness. I recently registered *VRa2* as the creator for Vermont

Recipes 2. Ordinarily, you should continue to use the same creator code for a new major version of an application, but I changed it for Vermont Recipes 2.0 because the application differs radically from Vermont Recipes 1.

The creator code, or signature, of a Macintosh application is based on old technology, but it still plays a role in Mac OS X versions through Leopard, particularly in older applications. Snow Leopard no longer honors creator codes when associating a document with an application, relying instead on newer techniques. You should nevertheless continue to register every creator code you use in your own applications with Apple, unless and until Apple stops accepting registrations. Go to <http://developer.apple.com/datatype/>. Limitations on which characters you may use are described on that Web page. You will receive an error message if your creator code is already taken by another developer. If registration is successful, Apple will notify you by e-mail.

10. Apple's *Information Property List Key Reference* indicates, in the "Recommended Keys for Cocoa Applications" section, that additional settings are required in every application's Info.plist file. The document-based application template does not provide them automatically, so you must add them manually. If you don't add them, your application will run correctly based on their default values or on entries you might add to the InfoPlist.strings file, discussed in Step 10. I nevertheless suggest that you include them in the Info.plist file based on Apple's admonition.

Order doesn't matter, so add them at the end of the list. To do this, first select the current last item, `NSPrincipalClass`, and click the Add (+) button to the right of that entry. A new entry is added below `NSPrincipalClass`, and a combo box opens displaying a list of available built-in settings. Repeat this procedure to add each of the following settings:

- a. Add the `CFBundleDisplayName` key and enter the value **Vermont Recipes**. You will localize this value, along with others, in Step 10.
- b. Add the `LSHasLocalizedDisplayName` key. For now, leave the checkbox in the Value column deselected to indicate that the value is 0 or false.
- c. Add the `NSHumanReadableCopyright` key and enter the value **Copyright © 2000–2009 Bill Cheeseman. All Rights Reserved**. The copyright symbol (the letter *c* in a circle) is Option-G on a U.S. keyboard. The dash in 2000–2009 should be an en dash (–) in correct typographical usage. On a U.S. keyboard, hold down Option and type a hyphen (-). This copyright string appears at the bottom of the application's About window and, in Snow Leopard where the Finder is now a Cocoa application, in a new Copyright field in the Finder's Get Info window. You will localize it in Step 10.

Many developers, including Apple, have traditionally inserted a linefeed character, using the escape sequence `\n` before the phrase, *All Rights Reserved*. This makes the copyright entry at the bottom of the About window look nicer. However, in Snow Leopard you shouldn't include a linefeed because it makes the Copyright field in the Finder's Get Info window look odd.

- d. Add the `NSAppleScriptEnabled` key. For now, leave its checkbox deselected. You will implement AppleScript support in Recipe 12.

- 11. *Information Property List Key Reference* states that the `CFBundleGetInfoString` key is obsolete in Snow Leopard, replaced by `NSHumanReadableCopyright`. In Snow Leopard, the Finder's Get Info window displays a copyright notice using the `NSHumanReadableCopyright` value you have already entered, instead of using `CFBundleGetInfoString` as it did in Leopard. However, Vermont Recipes 2 runs under Leopard as well as Snow Leopard, so you should add a `CFBundleGetInfoString` string.

Using the procedure described earlier, add the `CFBundleGetInfoString` key and enter the value **Vermont Recipes 2.0.0. Copyright © 2000–2009 Bill Cheeseman**. This differs from the `NSHumanReadableCopyright` value only because many developers, including Apple, have traditionally included the application name and version and omitted the phrase *All Rights Reserved*. You could use the same value as the `NSHumanReadableCopyright` entry instead, given that the application's name and version already appear elsewhere in the Get Info window. Whatever you do, you will localize it in Step 10.

- 12. The *Information Property List Key Reference* recommends that a universal binary like Vermont Recipes 2 should generally include the `LSExecutableArchitectures` key. However, the system automatically prefers the native architecture for the computer currently in use, and Vermont Recipes has no independent reason to overrule the default. For example, it does not need to run legacy plug-ins that might not be available in the current architecture. You therefore don't need to include this key or the related `LSRequiresNativeExecution` key.

All the other settings are correct, so you're done with the Info.plist Entries pane for now. In the next step, you will localize some of the values.

Step 10: Edit the InfoPlist.strings File

Another file in the Resources group in the Groups & Files pane is `InfoPlist.strings`. This file specifies the text of various application settings that are shown to the user in the running application, such as the name of the application, using the language of the computer's current locale. The English-language version of this file is located

in the English.lproj folder in the application package's Resources folder, and versions of the file in other languages are located in their respective .lproj folders.

By convention, all files that specify translation of text shown to the user in the running application end with the *strings* file extension. There can be many such files. The InfoPlist.strings file is special only in that it is used to provide localizations for system settings, mostly those in the Info.plist file that you just finished setting up. Another example is the Localizable.strings file that you will create in Step 11. Localization files like this are saved as simple property lists with a number of key-value pairs, often with comments to help localization contractors identify the use and purpose of each of the strings. Some of these strings are used, for example, in the application's About window and some by the Finder. All strings files should be saved in UTF-16 encoding.

The instructions here are for specifying several items in the InfoPlist.strings file that are considered mandatory or recommended according to Apple's *Information Property List Key Reference*.

1. Expand the Resources group and open InfoPlist.strings. The text editing window that opens is empty except for the comment */* Localized Versions of Info.plist keys */*.
2. Skip a line and type **CFBundleName = "Vermont Recipes";**. The trailing semicolon is vital. Without it, your application will fail to read the translations and strange things will happen. These errors can be very difficult to debug.

The value of the CFBundleName entry appears as the application's name both in its About window and as the title of the application menu. If you were to localize the application, you might decide to translate the word *Recipes* in *Vermont Recipes* to its equivalent in the local language.

Immediately following the CFBundleName entry, type a comment that will help your localization contractor to understand what this entry does and where in the application its text value appears. The comment is optional, but your contractor will appreciate the help. Type **/* Localized "short" application name in the menu bar and About window */**.

A localization contractor creates an almost identical file and places it in another lproj folder in the application package's Resources folder. The only difference between the two versions of the file is the language used to express the value in quotation marks. At runtime, the application checks the system's current locale and uses the translation from the appropriate lproj folder.

This is even true of the English.lproj folder when the application is running in the English locale. If the string you provide in the InfoPlist.strings file differs from the string you provide in the Info.plist file itself, your running application uses the string from the InfoPlist.strings file. You can see this happen if you put

a dummy name such as **New Hampshire Recipes** in the `CFBundleName` entry in `InfoPlist.strings` and build and run the application—the name of the application now appears as *New Hampshire Recipes* in the menu bar.

3. Skip a line and type **`CFBundleDisplayName = "Vermont Recipes";`** (including the trailing semicolon), and add this comment: **`/* Localized "long" application name in the Finder, etc. */`**. This entry is relevant only if your application supports use of a localized display name for the application package. If it does, you should also localize the `CFBundleName` entry. If it does not, remove the `CFBundleDisplayName` entry. The system overrides the name provided here if the user changes the name of the application in the Finder.
4. Skip a line and type **`NSHumanReadableCopyright = "Copyright © 2000–2009 Bill Cheeseman. All Rights Reserved.";`** followed by this comment: **`/* Localized copyright notice for About window */`**. This is identical to the copyright string you created in the `Info.plist` file in Step 9 because, of course, the book assumes you are developing in English. In Snow Leopard, it appears in the Finder's Get Info window as well as the application's About window. A localization contractor might want to translate this string for another locale.
5. Skip a line and type **`CFBundleGetInfoString = "Vermont Recipes 2.0.0. Copyright © 2000–2009 Bill Cheeseman.";`** or whatever you chose to use for the value of this key in Step 9, followed by this comment: **`/* Localized application name, version and copyright notice for Finder's Get Info window */`**. This string appears in the Finder's Get Info window in Leopard. A localization contractor might want to translate it for another locale (**Figure 1.11**).



FIGURE 1.11
The finished
InfoPlist.
strings file.

6. If you chose to include text as well as digits in the `CFBundleShortVersionString` key in Step 9, you may have to localize it, too. Skip a line and enter **`CFBundleShortVersionString = ,`** followed by the English version of the string that you used in Step 9, enclosed in quotation marks. Don't forget the trailing semicolon.

You're almost done with Recipe 1. For the sake of completeness, you'll add one more file to the project before concluding.

Step 11: Create a Localizable.strings File

A Localizable.strings file should be added to the project for each localization included in the application. It contains key-value pairs specifying string values for any localizable string you use in your code. Strings you name in an Interface Builder nib file can be localized using the nib file, so they don't have to be included in the Localizable.strings file.

Localizable.strings is the conventional name for the file if you have only one. The application can contain many similar files with other names and the *strings* extension. All *strings* files for a given language belong in that language's lproj folder. The files take the form of property lists containing key-value pairs in plain text, with comments to assist localization contractors. Here you will create a Localizable.strings file for the English.lproj folder.

Specifying strings in localizable form in your code is so easy that you should always do it. At any place in the application's code where you would normally use a statement like `@"this is a string"` to provide user-viewable text, you should instead use the `NSLocalizedString()` macro or, if you have *strings* files with names other than Localizable.strings, the `NSLocalizedStringFromTable()` macro. These macros are defined in the NSBundle header in the Foundation framework. They are conveniences that call NSBundle's `-localizedStringForKey:value:table:` method on the application's main bundle. In the `NSLocalizedString()` macro, you pass two parameters specifying the key and a comment string explaining what you're doing. The comment is not used at run time but exists only to support your localization contractor. It should be descriptive and should be repeated verbatim as a comment in the Localizable.strings file or a similar file. In the `NSLocalizedStringFromTable()` macro, you pass three parameters specifying the key, the name of the *strings* file in which the key and its paired value are found, and a comment string.

Typically, when your application is turned over to localization contractors, they add a new lproj folder for another language. Among other things, they place in it a copy of the Localizable.strings file and any other *strings* files you provide, using the same keys but translated string values. They also localize the application's nib files using Interface Builder, as well as provide localized images, sounds, and perhaps other resources. The localization contractors do not have to touch the application's code, because when you use these macros, Cocoa automatically uses the resources in the lproj folder corresponding to the language for which a particular computer is set up.

Whenever you insert the `NSLocalizedString()` macro or the `NSLocalizedStringFromTable()` macro into your code, you should also remember to return to the `Localizable.strings` file and similar files to provide a suitable key-value pair and comment to match. Fortunately, Apple provides the *genstrings* command-line tool to automate the process. Launch the Terminal application and type **genstrings** for help. A similar command-line tool, *ibtool*, is provided for localizing nib files. Later, you will learn how to include a script build phase in your project to run *genstrings* and fill out your `Localizable.strings` file automatically every time you build the project.

You don't yet have any strings to localize, other than those you've already placed in `InfoPlist.strings`. Nevertheless, create an empty `Localizable.strings` file now for possible later use.

1. With the Xcode project file active, choose **File > New File**.
2. In the first New File window, select the **Other** category and its **Empty File** template, and click **Next**.
3. In the next New File window, name the file `Localizable.strings`, choose the `English.lproj` folder as its location, designate **Vermont Recipes** as the destination project, select the **Vermont Recipes** target checkbox, and click **Finish**. In the Xcode project window, move the `Localizable.strings` file next to the `InfoPlist.strings` file in the **Resources Group** in the **Groups & Files** pane.
4. If you wish (this is not required), open the new, empty `Localizable.strings` file and type a descriptive heading such as `/* Localizable strings for Vermont Recipes 2.0.0 */`.

Step 12: Set the Project's Properties and Build Settings

The Vermont Recipes project builds and runs even if you make no changes to its properties and build settings, but you should get in the habit of setting every new project's properties and build settings. The options are voluminous, and they have an important impact on the user experience.

1. Begin by selecting the **Vermont Recipes** project, the top line in the Xcode project window's **Groups & Files** pane, and then click the **Info** button in the toolbar or use the contextual menu to choose **Get Info**. The Project "Vermont Recipes" Info window opens.
 - a. Click the **General** tab. Assuming you are using **Snow Leopard**, set the **Project Format** pop-up menu to **"Xcode 3.2-compatible."** You anticipate

no need to edit or build the project in Xcode 3.1 or older, so you might as well take advantage of whatever improvements have been implemented in the Xcode 3.2 project format (**Figure 1.12**).

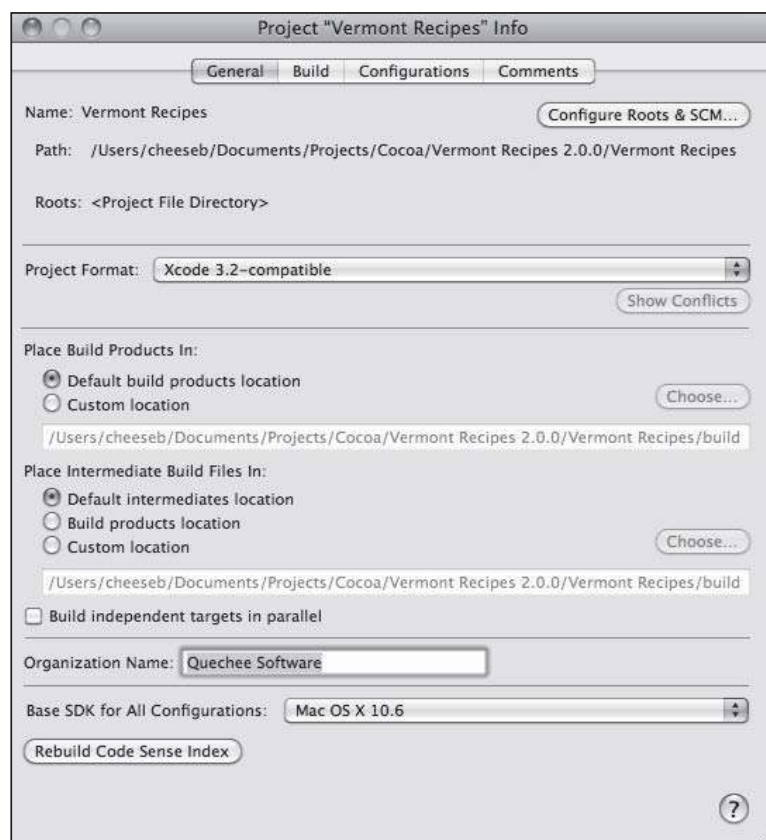


FIGURE 1.12
The General pane
of the Project
“Vermont Recipes”
Info window.

Documentation

Xcode Preferences and Build Settings

When you set up an Xcode project for any but the simplest product, it is essential to have at your fingertips a complete reference document for all available build settings. This is the document you need: *Xcode Build Setting Reference*. You should bookmark it in the Xcode documentation window

There are a number of related documents, ranging from overview to highly technical: *Working with Xcode Build Settings*, *Xcode Build System Guide*, *SDK Compatibility Guide*, *Runtime Configuration Guidelines*, *Information Property List Key Reference*, and *Xcode User Default Reference*.

You may also find it useful to read up on related developer tools, particularly the compiler. For information about the latest changes, read *GCC 4 Release Notes* and *LLVM-GCC Release Notes*. There are also a number of very lengthy GNU documents about the compilers and other tools.

Read *Uniform Type Identifiers Overview* for details regarding UTIs.

Leave the build products and intermediate build files locations at the default values, specifying that their locations be controlled by the general Xcode preferences you set in Step 3. You can set different locations for a specific project here, but there is no reason to do so for Vermont Recipes.

The “Base SDK for All Configurations” setting is Mac OS X 10.6 (Snow Leopard) by default. With this setting, your application can include new features implemented for the first time in Snow Leopard. Vermont Recipes 2 is going to include some features that are new in Snow Leopard, so leave it set as is. It will mean little to you now, but this setting controls *SDKROOT*. Setting it to the path of a specific Mac OS X SDK, such as Mac OS X v10.6, enables Xcode to search for declarations in frameworks, headers, and libraries in that SDK, and it sets the *MAC_OS_X_VERSION_MAX_ALLOWED* preprocessor macro to the same version of the operating system.

If you ever get the feeling that the project’s index has gotten out of whack, come back to the General pane and click the Rebuild Code Sense Index button at the bottom.

- b. Click the Build tab. Don’t be discouraged by the long and bewildering list of build settings, and above all don’t change any settings here until you know what you’re doing. You will eventually understand most if not all of these settings, but you can do a lot of damage if you change them inappropriately (**Figure 1.13**).

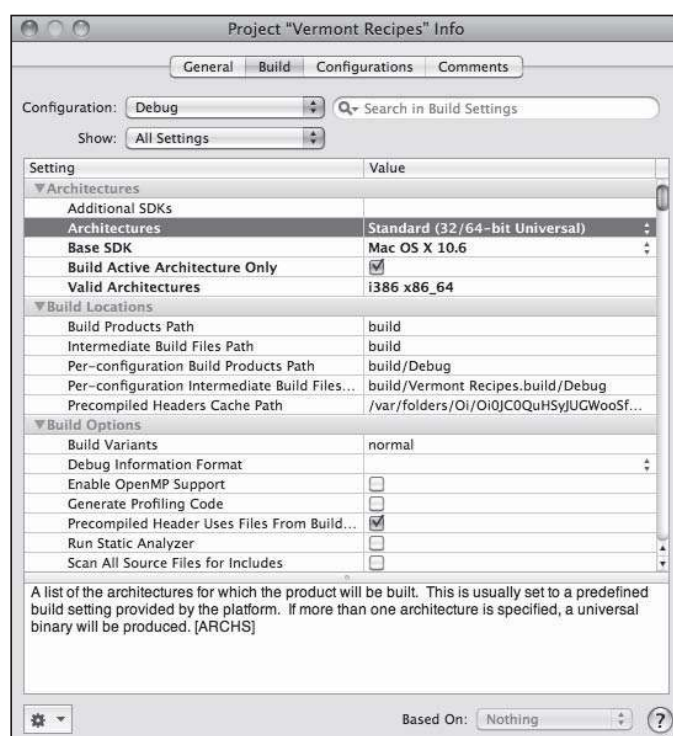


FIGURE 1.13 The Build pane of the Project “Vermont Recipes” Info window.

Settings in the project Build pane are inherited by all of the project’s target build settings, which you will explore shortly. It is therefore convenient to

change any settings here that should be the same for every target in the project. Later, you can make any target-specific changes in the target's Build pane.

By default, the titles and values in the Build pane are descriptive. A contextual menu lets you change the Setting column of all rows at once to show the actual name and to change the Value column to the actual definition. For example, the setting called Base SDK, which you just saw is set to Mac OS X 10.6, becomes SDKROOT when you choose to display its name. This is useful when you're writing a script build phase, where you must use setting names, not their descriptive titles. The value for Base SDK is shown as Mac OS X 10.6, while the definition is shown as `macosx10.6`. The definition is especially important when you use variables to set the value, such as a search path, because when definitions are showing, you can see the variables that determine the values.

With the Configuration pop-up menu at the top of the Build pane, you can change the settings separately for the Debug and Release configurations. These and other configurations are created in the Configurations pane. It is often important to use different settings in the Debug configuration from what you use for the Release configuration.

To quickly see what settings have been changed from their default values in a particular configuration, choose Settings Defined at This Level from the Show pop-up menu. You can also create and show custom settings. When looking for a particular setting or group of similarly named settings in the All Settings view, use the search field if you remember the name.

When any setting is selected, a full description appears in the pane at the bottom.

For now, change only one of the project's build settings. This change should apply to both the Debug and the Release configurations, so choose All Configurations from the Configuration pop-up menu. Choose All Settings from the Show pop-up menu; then scroll down to the Deployment section and select the Mac OS X Deployment Target setting. It is set by default to the current version of the operating system, Mac OS X 10.v6, but you want Vermont Recipes 2 to run under Leopard as well as Snow Leopard. Use the pop-up menu beside this setting to choose **Mac OS X 10.5**. Now, Vermont Recipes will refuse to run under Mac OS X v10.4 and earlier, but it will load and run under Leopard and Snow Leopard. You have set the deployment version, or `MAC_OS_X_VERSION_MIN_REQUIRED` preprocessor macro. You will learn later what you have to do to ensure that the features in Vermont Recipes 2 that aren't supported under Leopard don't cause problems when you run the application under Leopard.

You will come back and change a number of other settings later. For now, be aware that, if you're building on an Intel Mac, the built application will not run on a PowerPC Mac. You'll fix that along with other things later.

Close the project info window.

2. Next, select the Vermont Recipes target by expanding the Target group in the Groups & Files pane, and then click the Info button in the toolbar. The Target “Vermont Recipes” Info window opens. Ignore the General and Rules panes for now.
 - a. Select the Build tab, and you see a pane that is identical to the project Build pane. This is where you change settings that differ from the project setting for this particular target. Otherwise, all targets inherit the project's build settings.
 - b. Select the Properties tab. You should recognize all of the settings in this pane, because you just set or reviewed them in Step 9. This pane is nothing more than a window into the application's Info.plist file. You can make changes in either the Info.plist file or the Properties pane. The Properties pane is useful only in that it makes it easy to see in one place, in an organized layout, the most important settings governing the application's interaction with the outside world (**Figure 1.14**).

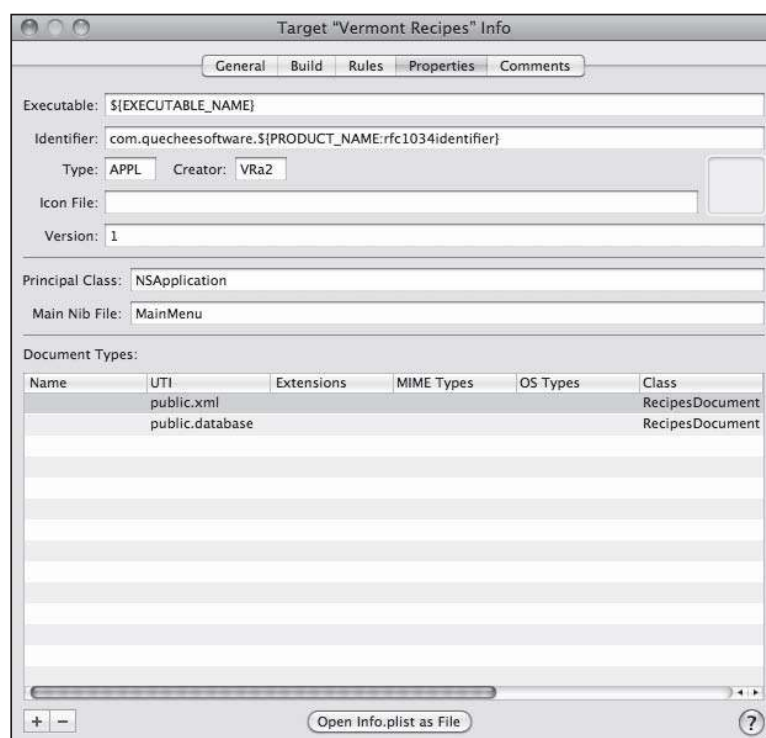


FIGURE 1.14 The Properties pane of the Target “Vermont Recipes” Info window.

Click the “Open Info.plist as File” button at the bottom of the Properties pane. The Vermont_Recipes-Info.plist file opens. You see the same settings as well as the others that you worked with in Step 9.

Close the Target “Vermont Recipes” Info window and the Info.plist window.

Step 13: Build and Run the Application

To celebrate the conclusion of Recipe 1, build and run your new application. It won't do much yet, but you will be encouraged to see that you have made real progress.

Building and running the application couldn't be easier. Just click the Build and Run icon in the Xcode project window's toolbar. If you left some files unsaved, Xcode immediately reminds you to save them. Click Save All and wait a few moments.

Suddenly, the Vermont Recipes database document window opens, its menu bar appears at the top of the screen, and a generic application icon appears in the Dock.

Many of the menu items already work. Be sure to choose About Vermont Recipes in the application menu to review your handiwork in the version number, the scrolling text view, and the copyright notice at the bottom.

Congratulations! You've just become a Cocoa developer.

Step 14: Save and Archive the Project

Make it a habit to close the project and set aside a copy for safekeeping at the end of each recipe. I find it useful to save compressed copies of my work in process from time to time. That way, I have something recent to go back to if I make a mess of the next set of changes. You really should use a Source Control Management (SCM) system to do this for you, but for now use the Finder.

1. In the running Vermont Recipes application, from the application menu, choose Quit.
2. Click the close box in the title bar of the main Xcode project window. Quit Xcode if you like.
3. In the Finder, open the Vermont Recipes project folder and drag the build folder in it to the Trash. The build folder is where Xcode placed the built Debug version of your application and a lot of related files when you built the application. You don't need to waste space by saving them, because they can easily be regenerated from the project files.
4. Navigate up one level, to the Vermont Recipes 2.0.0 folder if that's your project's parent folder. Select the Vermont Recipes subfolder and use the Finder contextual menu's Compress "Vermont Recipes" command to create a zip file. To distinguish it from backups of later versions of the project, rename it from *Vermont Recipes.zip* to something like **Vermont Recipes 2.0.0 - Recipe 1.zip**.

Save a copy of the zip file on another disk, burn it to a CD or DVD for safe-keeping, or, if you have a MobileMe account, save it on your remote iDisk in case your house burns down. The working Vermont Recipes project folder remains in place, ready for you to start work on Recipe 2, coming up next.

Conclusion

You've made a lot of progress on your foray into the world of Macintosh Cocoa application development. You have created a new project. You have populated it with code files to run the application and create documents, nib files to display and control the graphical user interface, and other resource files that define the behavior and appearance of the application. You have even built and run the application.

But you still have a lot to do to turn the application into a useful tool for your users. In the next recipe, you will set the stage by adding the visual props needed to form the background for all the controls and menu items your users will need to do real work.

Documentation

Xcode

Apple offers several introductory and summary documents regarding Xcode and related developer tools. Before moving on to more detailed and technical documentation, read: *Getting Started with Tools & Languages*, the "Xcode" section of Appendix C of *Mac OS X Technology Overview*, and *A Tour of Xcode*.

For documents focusing on new features in the latest versions of Xcode, read *What's New in Xcode*, *Xcode Release Notes*, and the "Xcode 3.0" section of *What's New In Mac OS X*.

It is always important to keep up with the release notes for new versions of the developer tools. A complete collection of release notes for Xcode 3.x is available in Xcode's Help menu.

The *Xcode Installation Guide* gives you information about setting up Xcode.

For more detailed and technical information, read *Xcode Project Management Guide* and *Xcode Workspace Guide*.

Additional technical publications focusing on Xcode preferences, build settings, and build phases are cited in Recipe 3.

Design and Build the GUI Using Interface Builder

The name of the Interface Builder application, commonly referred to as IB, suggests that it is a tool for building the graphical user interface (GUI) of an application. It is that, and much more. It is an interface design tool you use to create the appearance of your application's windows and views. It is a testing tool you use to run your user interface even before you write any code. And it is a generator of archives of interface objects such as windows, views, and menus, ready to be unarchived and connected for use by your application at run time. Using Interface Builder, you set many default values and behaviors of your application's user interface elements without writing a single line of code.

HIGHLIGHTS

Creating and setting up an Interface Builder nib file

Creating a toolbar and toolbar items

Creating a split view

Using Springs and Struts to control a view's resizing behavior

Creating a table view

Creating a tab view

Creating a drawer

In the course of designing the GUI for the Vermont Recipes application, you will see that the nib files you create in Interface Builder contain a significant amount of information and functionality relating to the application's user interface, saving you from having to write an immense amount of code. With Interface Builder, you draw and prebuild Cocoa window, view, and other classes and objects, as well as connections between them and their methods; and you archive them for loading directly into your application at run time. This is a different approach from that taken by many GUI design utilities for other platforms, which, for example, might generate textual code snippets to be compiled later when you build the application. It also differs from the approach taken by ResEdit, used in Mac OS 9 and earlier to create encoded layout templates as application resources.

While it is possible to build the GUI of a Cocoa application from code without using Interface Builder, it is hard to imagine a reason for undertaking such a tedious and error-prone task. Apple specifically recommends against it in the Nib Files topic in the *Resource Programming Guide*.

You can examine and edit some of the content of the nib files of older Macintosh applications, even without access to their source code. Starting with Leopard, however, Apple gives developers the ability to compile nib files into a format that cannot be read or altered by humans without access to the original. There is also a new kind of nib file with a different file extension, *.xib*, but most developers continue to call them *nib files*.

When you reach the point of writing code for the application, you will discover one reason why the nib file's internal information is important. Many seemingly essential items are omitted from the Xcode source files. For example, you declare some instance variables in code, yet you write no code telling the application which controls the instance variables point to. Similarly, you implement some methods in code, yet you write no code telling the application which action methods the specific controls should invoke.

Interface Builder's nib files supply the omitted information. As you design the GUI for the application, you typically create outlets and actions by writing code in Xcode, and then connect them to appropriate objects using Interface Builder's intuitive graphical interface. Outlets are instance variables declared in code that you connect to objects in the nib file. You use Interface Builder to draw a connection from an object containing an outlet to an associated object such as a control, and then specify which of the object's instance variables to connect. Similarly, actions are methods you implement in code. Again, you use Interface Builder to draw a connection from a control to the target object for the action message, and then specify which of the target's action methods to invoke when the user clicks the control. Cocoa automatically invokes the correct method at run time whenever the user clicks the control, with little or no further coding on your part. Read the "Outlets and Actions" sidebar for more information. You also use Interface Builder to set the default values and attributes of many controls. Newer technologies such as Cocoa Bindings, also supported by Interface Builder, make the development process even easier.

As you use Interface Builder to create outlets, actions, connections, bindings, and other features of the GUI, the information is stored in the nib file. The nib file forms an integral part of the application, and the information in it is used at run time to pull everything together into a smoothly functioning whole.

It is generally advisable to create a separate nib file for each major window in an application. You can also make separate nib files for views, such as the panes in a complex window. This makes it possible for your application to load each of them lazily, when needed. Your application tends to start up more quickly because it has

Outlets and Actions

You see in this recipe that Interface Builder relies to some extent on an electrical metaphor. Objects have outlets, and you plug other objects into them by stringing wires between the outlets and the objects to be plugged into them. Interface Builder at one time even used an electric outlet symbol to identify outlets.

In programmer's terms, an outlet is an instance variable or property declared in a class header. Typically, a class such as a document class or a window controller class declares numerous outlets identifying other objects with which it needs to communicate, including not only controls but also any kind of object. The instance variable or property is simply a pointer to the object. In code, you identify an outlet by adding the term `IBOutlet` to the beginning of its declaration, before the type declaration. This tells Interface Builder that it is an outlet, so that Interface Builder can update itself to remain synchronized with your code.

You use Interface Builder to wire user interface objects with a sort of control circuit, connecting each control object to a specific action method in a target object, to be invoked when the user makes use of the control. This implements the Target-Action design pattern, another design pattern that lies at the heart of Cocoa. Typically, a control is connected to an action method implemented in a window controller object. In code, you declare the return type of the action method as `IBAction`. This is equivalent to `void` as far as Xcode knows, but it tells Interface Builder that it is an action method so that Interface Builder can keep itself synchronized with your code.

In this way, you can divorce your user interface code from your substantive code with Interface Builder to a much greater extent than is true of other programming environments. One of Interface Builder's functions is to provide information needed at run time about what is connected to what, so you don't have to lock this information into your code. Interface Builder does much more than its name suggests.

less data to load at launch time, and opening auxiliary windows proceeds quickly if each is in its own nib file. A common exception to this rule of thumb is to combine an application's main menu and a window that is always opened at launch time into one nib file, because they both have to be loaded at launch time anyway. Objects that are related to the main window in the nib file should be included in it, such as the drawer and its associated drawer content view in the parent window's nib file.

In this recipe, you learn how to use Interface Builder 3.2 in Snow Leopard to design and build the beginnings of a graphical user interface for the application's main document window.

Step 1: Explore and Revise the Document Window's Nib File

It is usually convenient to design and build the essential elements of the user interface for the application's main document window early in the development cycle. This gives you an opportunity to begin testing your initial application specification by putting it on the screen in the same form that the user of the final application will see. Interface Builder's ability to run the interface in interactive test mode before you have written much code is often useful in verifying your assumptions about usability. At this early stage, using Interface Builder as a prototyping tool, you can easily make adjustments and fine-tune the GUI.

1. Start by opening the Vermont Recipes 2.0.0 folder in which you saved the project folder at the end of Recipe 1. Leave the compressed project folder you set aside at that time where it is, and open the working Vermont Recipes subfolder. In it, double-click the Vermont Recipes.xcodeproj file to launch Xcode and open the project window.

You have a housekeeping matter to take care of first. Recall from Recipe 1 that you decided to increment the application's `CFBundleVersion` setting at the beginning of each recipe. Open the `Vermont_Recipes-Info.plist` file in the Resources group of the Groups & Files pane, change the value of the `CFBundleVersion` key from 1 to 2, and save and close the file. When you open the About window at the conclusion of this recipe, you will see the application's version displayed as *2.0.0 (2)*.

2. Now you're ready to look at a nib file. Although it is possible to work on nib files in Interface Builder while the project is not open in Xcode, the better practice is to have the project open in Xcode at the same time. You can make adjustments to the code while you are working on the user interface, and Xcode and Interface Builder work together behind the scenes to keep your work in both applications synchronized.

In Xcode, expand the Resources group in the Groups & Files pane and double-click `RecipesWindow.xib`. Interface Builder launches and opens the nib file. Alternatively, launch Interface Builder and use it to open the nib file.

Two windows and two palettes appear. The nib file's document window is titled `RecipesWindow.xib - English` (**Figure 2.1**). It is your main point of entry to the nib file. It initially contains icons labeled File's Owner, First Responder, Application, and Window.



FIGURE 2.1 The RecipesWindow.xib document window in icon view mode.

The main recipes document’s window destined for your application is open on the screen nearby. This window, like view and menu objects when they are open for editing in Interface Builder, is called the *design surface*. Named Window in this case, it contains a text field with the sentence “Your document contents here.”

The Library window is a palette that is visible while Interface Builder is the active application (**Figure 2.2**). From it, you drag various views, controls, and other objects to the design surface and to the nib file’s document window. If you ever close the Library window, you can reopen it by choosing Tools > Library.



FIGURE 2.2 The Library window.

Finally, the Inspector window is another palette. Like the Library window, it is hidden whenever you make another application active. It contains a toolbar

enabling you to select one of several different inspectors. If you close it, you can reopen it to show its most recent contents by clicking the Information button in the document window's toolbar or by choosing Tools > Inspector. You can show a specific inspector by choosing, for example, Tools > Size Inspector. You use the various inspectors for many purposes, including to set a control's initial attributes, to set up its Core Animation effects, to set its size and resizing behavior, to set its bindings or connections, and to define its identity by, for example, designating a specific class or custom subclass that defines it.

The inspector's contents change automatically when you select different items in the document window or the design surface, enabling you to set values for a particular window, view, control, or other object. Try that now. Click the text field reading "Your document contents here" in the design surface, and see the title of the inspector change to something like Text Field Connections. Then click the Window icon in the document window or the title bar of the window design surface, and see the title of the inspector change to something like Window Connections.

With the document window still selected, click the Attributes button at the left end of the inspector's toolbar. You are now looking at the Window Attributes inspector (**Figure 2.3**). The Attributes inspector contains a unique group of settings you can use to configure the most important attributes or properties of a specific view or control. Most of these settings correspond to an instance variable or method in the corresponding Cocoa class. In effect, when you configure the Attributes inspector, you are assigning values to instance variables in the object. Whenever you're creating a new object in Interface Builder, it is important to examine the Attributes inspector to determine whether any of the default attributes should be changed.



FIGURE 2.3 The Window Attributes inspector.

The label given to an attribute in the inspector does not always provide a clear understanding of the behaviors that the attribute controls. Fortunately, the Attributes inspector includes help tags. For example, in the Window Attributes inspector, the Visible At Launch checkbox has caused confusion for years. Now, however, if you pause the pointer over the label, a help tag appears explaining

what it does: “Indicates whether the window is (immediately) visible when the NIB is loaded.” In other words, setting this attribute causes the window to appear only when this particular nib file loads, not when the application launches, as some once thought. Many nib files aren’t loaded until the user asks the application to open the window. Leaving this attribute checkbox deselected lets you preload the nib file without causing the window to appear immediately, in case doing this is important in your application.

Other inspectors will be discussed in more detail later.

3. Explore the default objects that Interface Builder has set up for you. The full meaning of what you find will become clear later.

Start by clicking the window’s title bar in the design surface to select it. In the Window inspector, click the Connections button in the toolbar (it’s the right-pointing arrow in a circle; its name appears in a help tag if you pause the pointer over it) or choose Tools > Connections Inspector. You see in the inspector three lists of outlets, received actions, and referencing outlets. Two of them, the **delegate** outlet and the **window** referencing outlet, are highlighted, and both indicate that they are connected to the File’s Owner proxy. The template you chose when you created the Vermont Recipes project provided this connection.

Move the pointer over either of the selected outlets and look at the recipes nib file’s document window. You see that the File’s Owner proxy is highlighted when you roll the mouse over the connection.

In the inspector, click the Clear (x) button before the File’s Owner reference connected to the **delegate** outlet. You have just broken the **delegate** outlet connection.

Click the now-empty circle located to the right of the **delegate** outlet and drag to the File’s Owner proxy in the document window (**Figure 2.4**). When you release the mouse button, the inspector updates to show that you have reconnected the **delegate** outlet. The delegate connection means that, at run time, the window object delegates some of its functionality to your recipes nib file’s owner. You will learn in a moment about the object that is represented by the File’s Owner proxy. Delegation is an important Cocoa design pattern that you will learn more about later.

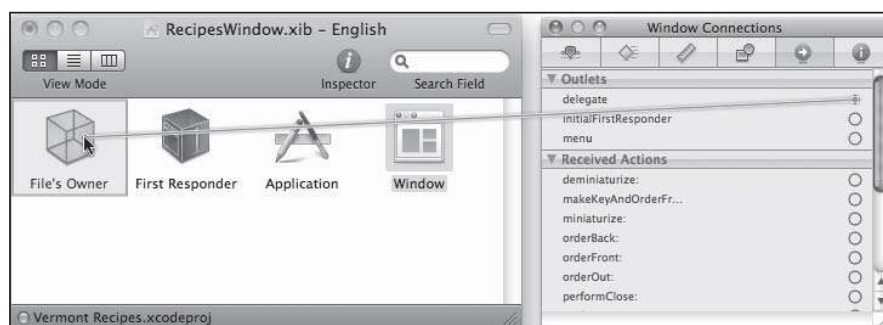


FIGURE 2.4 Connecting the window’s delegate outlet to the File’s Owner proxy.

File's Owner

The File's Owner icon in an Interface Builder nib file is a proxy for whatever object owns the nib file.

A proxy is used because the nib file's owner cannot be instantiated in the nib file. It's a chicken-and-egg problem. The running application must load the nib file. But for this to happen, the File's Owner must already exist in the running application. You must therefore create it programmatically.

Every application has a main nib file whose owner is the application object, `NSApplication`. The `NSMainNibFile` key in the application's `Info.plist` file identifies the main nib file of a Cocoa application. The application loads its main nib file at launch time. In a typical application, the main nib file is the `MainMenu` nib file, which is responsible for the application's menu bar and potentially for other interface objects.

As you learned in Recipe 1, a new document is instantiated programmatically at run time using hard-wired Cocoa code when the application is launched or every time the user calls for a new document. You write your document subclass in either of two ways. In a simple application, the document loads the nib file directly, in which case the document is the File's Owner. In a more complex application, the document instantiates a window controller, which in turn loads the nib file. In that case, the window controller is the File's Owner. Either way, the owner of the nib file must exist in memory as an instantiated object before it can load the nib file. The owner is in this sense external to the nib file that archives the window.

There must be some means of communication between the File's Owner and other objects, such as a window, instantiated in the nib file. The File's Owner stands in for the owning object for this purpose. Its icon is used when you draw the necessary connections between it and other objects while designing the user interface. When the nib file is loaded at run time, Cocoa takes care of instantiating the connections to the real file's owner object, using the information regarding connections to the File's Owner proxy that it finds in the nib file.

4. Click the File's Owner proxy in the document window. The Window Connections inspector automatically becomes the My Document Connections inspector. You never have to close the inspector and reopen it to see information about another object.

If you're alert, you just realized that you've discovered yet another place where you must change references to the document formerly known as `MyDocument`. Interface Builder is very smart about automatically keeping up with changes to your code in the Xcode project, but it can't be sure that your intent, when you changed the name of the class in the header file or even when you changed the name of the header file itself, was to change the identity of the nib file to match.

You learned in Recipe 1 that complex multidocument, multiwindow applications should use a custom subclass of `NSWindowController` to handle communications between the document and its window. To accommodate that requirement, you created the `RecipesWindowController` class header and implementation files. The document-based application template you selected when you created the Vermont Recipes project made the `MyDocument` class the owner of the nib file, so you must now change its owner to the `RecipesWindowController` class. You already named the nib file correctly, in anticipation of this change, in Recipe 1.

To edit the nib file's reference to the document, click the Identity button in the Inspector's toolbar (it's the letter *i* in a circle). You see at the top of the My Document Identity inspector that Interface Builder still thinks the class of the nib file's owner is `MyDocument`. To change it, pull down the menu, scroll up or down as needed, and select `RecipesWindowController`. Alternatively, type **RecipesW** into the text field and watch Interface Builder autocomplete the name as soon as it knows you aren't typing `RecipesDocument`. The name of the inspector immediately changes to identify it as the Recipes Window Controller Identity inspector (**Figure 2.5**).



FIGURE 2.5 The Recipes Window Controller Identity inspector.

The nib file already knew about your new `RecipesWindowController` class because Interface Builder does a good job of staying synchronized with the Xcode project. If you ever find that Interface Builder has gotten out of sync with the project, choose **File > Reload All Class Files**; or choose **File > Read Class Files** and, in the Open panel, navigate to the Vermont Recipes project folder, select the missing header file, and click **Open**.

You will learn more about the concept of a nib file's owner in Cocoa in some detail later. For now, think of it simply as the object that loads the nib file at run time. In this case, when the user launches the finished Vermont Recipes application, the application tells the main recipes document to open and the

RecipesWindowController object to load this nib file to set up the user interface for the document's window.

5. There are some additional features in the document window that you should explore now. Start typing **RecipesWindowController** into the search field. Before you can finish typing, Interface Builder autocompletes the name. The window switches from icon view to outline view. The File's Owner proxy is the only item showing, with the RecipesWindowController class specified as the file's owner. Click the Clear (x) button in the search field to remove the search text, and the window reverts to icon view.

Using the segmented control at the left end of the document window's toolbar, you can work in outline view mode or browser view mode all the time. This is often more useful than icon view mode, especially if your nib file fills with dozens of views and controls. It also makes your job much easier when you need to see the hierarchy of objects you will create in Interface Builder to complete your application's user interface.

To see how this works, choose outline view mode and click the disclosure triangle to the left of the Window object. The entry expands to show the Content View that every window contains. Expand the Content View entry, and see the one static text object currently visible in the recipes document's main window. Click the Static Text entry, and see the static text field's cell. Later, after you add more controls, you will see them in the expanded outline as well (**Figure 2.6**).

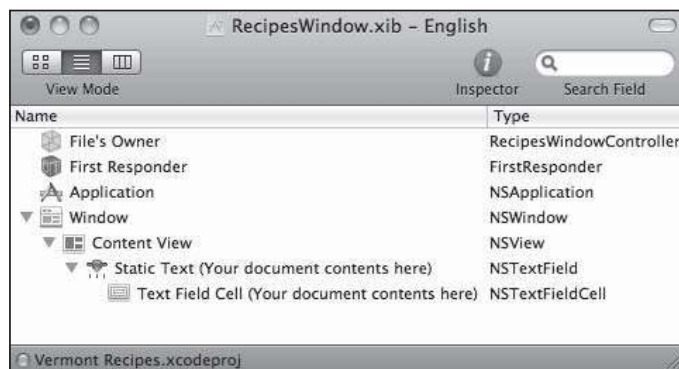


FIGURE 2.6 The RecipesWindow.xib document window in outline view mode.

The icons or entries at the top level of the nib file's document window represent objects that Interface Builder instantiates and archives in the nib file. When you see an object at the top level, you know that you do not have to create and initialize the object in your code. Instead, it is instantiated automatically when your application unarchives and loads the nib file. By the same token, since you don't see a window controller icon at the top level of the document window, you know that Interface Builder does not instantiate it, and that you must create and initialize it at an appropriate time in your code. You already arranged to do that when you added the `-makeWindowControllers` method to the `RecipesDocument.m` implementation file in Recipe 1.

Cocoa Views

To learn more about Cocoa views in general, read the *View Programming Guide for Cocoa*.

To learn about the specific views described in this recipe, read the following Apple documents: *Toolbar Programming Topics for Cocoa*, *Tab Views*, *Drawers and Application Menu*, and *Pop-up List Programming Topics for Cocoa*. With respect to split views, read the AppKit Release Notes for Leopard and Snow Leopard, both of which contain a wealth of information about split views that has not yet made it into other documentation.

Two Cocoa AppKit classes are discussed in some depth in this recipe. For more information about them, read the *NSSplitView Class Reference* document and the *NSDrawer Class Reference* document.

You can add instantiated objects, such as new windows and views, to the nib file by dragging them from Interface Builder's Library window. In the next series of steps, you learn how to use the Library window to add several views to the recipes document's existing window. In this step, you leave each of these views empty, setting the stage for the controls you'll begin adding in a later recipe.

Step 2: Add a Toolbar

A toolbar across the top of a window has become a popular user interface device to make an application's most frequently used commands easily available to the user. These commands typically duplicate menu commands, but they can be executed with a single click in the toolbar. The toolbar also serves to remind users of the application's principal features. Since different users may take different approaches to any application, Cocoa makes it easy for you as a developer to let the user customize the toolbar.

As an experienced user of Macintosh applications yourself, you know almost without having to think about it that the main recipes window should include a toolbar. The toolbar houses a search field so that the user can find recipes and ingredients. It includes an Information button to open and close the drawer that you will add to the document window shortly. It will need a print button. Others will undoubtedly occur to you.

Add a toolbar to the window as a first exercise in adding a view to a window using Interface Builder. Defer populating the toolbar with custom toolbar items until Step 7, after you have added several other views to the window.

Before proceeding, get rid of the text field reading “Your document contents here.” Click the text field to select it, and press the Delete key.

1. Start the process of adding a toolbar by examining the Library window. It holds a lot of information, and it provides several devices to help you find what you want.

The row of tabs at the top lets you choose classes such as `NSAlert`. You can instantiate an object based on any of these classes by dragging it into the design surface. You can also instantiate Media, such as an Information button image, by dragging an image into the design surface. Select the Classes and Media tabs now to take a look.

2. Select the Objects tab. You see a long list of objects below it, starting with a menu object and ending, after a lot of scrolling, with a QuickTime Capture View object. Interface Builder provides a wealth of prebuilt view and control objects, but these riches can be a little overwhelming.
3. Fortunately, the classes are broken down into categories. Use the pop-up menu at the top of the Library window to choose Library > Cocoa > Application > Toolbars. Alternatively, drag the horizontal divider downward and the pop-up menu becomes an outline view in which you can see several entries at once. You now see a much more manageable short list of view objects in the center pane, including the Toolbar object itself and a number of toolbar item objects. Select the Toolbar object. Both in the selected view and at the bottom of the Library window you see a short description making it clear that this is the view you want. When you become more familiar with what is available, you can use the contextual menu or the action button at the bottom of the window to reduce the size and amount of information shown in the window, while still seeing the full description in the pane at the bottom.
4. Drag a Toolbar object out of the Library window and drop it near the top of the design surface. Don't be finicky about exactly where you drop it, because the Toolbar already knows that it must be located at the top of the window and cover the window's full width. You'll see in the next step that, even with views that don't have a predetermined location or size, it's easier to set a view's position and size correctly by dragging and resizing it after it has been added to the design surface.

As you drag the small Toolbar image into the window, it grows into a full-size toolbar, and you see that it is already populated with Colors, Fonts, Print, and Customize buttons (**Figure 2.7**).



FIGURE 2.7 The design surface after adding a toolbar.

5. By dropping the Toolbar object into the design surface, you began to create the user interface element hierarchy that is typical of every Macintosh window. The details and structure of the hierarchy are not readily apparent, however, because many views in a window are hidden or obscured.

To see the full hierarchy and to gain access to each subview for editing, go to the nib file's document window and choose the outline view mode. Expand the Window entry by clicking its disclosure triangle. You see that the window contains two views, the new Toolbar object you just added and the standard Content View object provided by the template. Click the Toolbar's disclosure triangle to expand it, and you see seven toolbar item objects, including the four buttons you saw when you dropped the toolbar into the window and the three invisible separator and space items needed to position them in the toolbar. If you prefer the browser view, open it and see how easily you can select a view object at each level and immediately see all of its children.

The Toolbar object is typical of the view objects that Interface Builder provides in the Objects pane of the Library window. You drag a view object into the design surface, and you see that it is instantiated with many of its features already in place.

To developers encountering Interface Builder for the first time, this often feels uncomfortably like magic. They may instinctively feel that they would prefer to write code to create a user interface. Resist that impulse! When Interface Builder archives these objects in a nib file for you, it is as if Interface Builder has written the code, compiled it, and stored it in the nib file for you. Interface Builder doesn't make spelling mistakes, and it knows most or all of the implementation details that a particular view requires. With Interface Builder, you can build views for your windows much more quickly than you can with hand-wrought code.

6. You are now ready to use the Cocoa Simulator, a feature of Interface Builder that lets you run the user interface features destined for your application without having to write any code. The Simulator is, of course, limited to those features that you have built into the nib file. With that qualification, the Simulator puts your window up on the screen exactly as it will look when you run

the finished application, so that you can exercise the built-in behaviors of the window and its views. As you will see in the next step, it even lets you exercise behaviors that you have added and customized in Interface Builder. Allowing you to test your interface objects while you're designing them is one of Interface Builder's most valuable features. This is not about testing their correctness, but about testing the workability and user friendliness of your design.

Run the Cocoa Simulator now. Choose File > Simulate Interface. Interface Builder's windows disappear, and in their place you see your recipes document window, complete with a title bar and its standard buttons such as the zoom button, the new toolbar with its built-in buttons, and the resize control at the bottom-right corner of the window.

Remarkably, almost all of these user interface elements work. Try them out. For example, click the minimize button and watch the window minimize to the Dock. Double-click its icon in the Dock, and watch the window re-form on the screen. Drag the resize control around and watch the window resize. While you're resizing the window, see that the toolbar automatically changes width to match the window's width, and the Customize button continues to hug the right end of the toolbar as the flexible space expands and contracts to hold it there. Click the toolbar button in the window's title bar and watch the toolbar shrink to invisibility, and then click it again and watch the toolbar expand to full size. Click the Colors and Fonts toolbar buttons, and watch the system Colors and Fonts panels open.

Click the Customize button, and see that Interface Builder has given you a fully functional sheet with which the user of your application can customize the toolbar. Use the pop-up menu to set the toolbar to display icons only or text only. Select the Use Small Size checkbox. Delete the Fonts button by dragging it from the toolbar to the desktop, and watch it disappear in a puff of smoke. Now click the Done button and see that your changes have taken effect. Click the Customize button again and drag the default set into the toolbar; then click Done to see that the original configuration has been restored.

7. When you've finished playing with the recipes document window and its toolbar, choose Cocoa Simulator > Quit Cocoa Simulator. Your document window closes, and the Interface Builder windows and palettes reappear.

The Toolbar object provided by Interface Builder requires less configuration than many views. Basically, you only have to add toolbar items, which you will learn how to do in Step 7.

Step 3: Add a Vertical Split View

The Vermont Recipes application specification tells you that the application uses Core Data to manage a database of recipes. A traditional user interface for databases on the Macintosh is what some have called a master-master-detail view. A well-known example is iTunes. Its main window has a vertical pane down the left side holding categories and subcategories in an outline view. When you select the Music Library, a pane across the top of the right pane holds a table view where you can select genres, artists, and albums. A third pane below it contains detailed information that changes to reflect the artist or other item selected in the top pane. You can move the divider between any two panes to the left or right or up and down to change the relative sizes of the panes. In this recipe, you adopt the master-master-detail model for the main recipes window.

For now, create the split views without content. Subviews and controls for the two panes will come later, starting with a Tab View object in Step 5.

1. Use the pop-up menu (or the outline view, if you expanded it) at the top of Interface Builder's Library window to select Library > Cocoa > Views & Cells > Layout Views. You see a short list of views that have in common the ability to lay out data in a visual organization within a window. Select Vertical Split View. You see a short description both in the selected view and at the bottom of the window making it clear that this is the appropriate view in which to set up a master-master-detail view.
2. Drag a Vertical Split View object out of the Library window and drop it anywhere in the document window beneath the toolbar. Its graphical representation grows somewhat larger and acquires handles. The Toolbar object you added in Step 2 didn't have handles because there was no need to reposition or resize it. Shortly, you will reposition and resize the Vertical Split View (**Figure 2.8**).

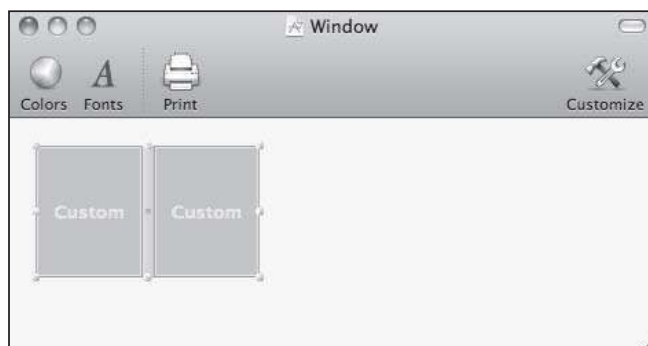


FIGURE 2.8 The document window after adding a split view.

3. To see the full hierarchy of the Vertical Split View object and to gain access to each subview for editing, go to the document window and choose the outline view mode. Fully expand the Window entry and all of its subsidiary entries, and

you see that it contains, from top to bottom, the standard Content View object provided by the template for every window, the Split View object you just dragged into the window, and two Custom View objects that Interface Builder provides.

4. To edit the attributes of the split view, click the Split View entry in the document window's outline. The inspector on the right side of your screen changes to show information about the split view. Click the Split View Attributes inspector button in the inspector's toolbar or choose Tools > Attributes Inspector. You see from the Style pop-up menu in the Attributes inspector that by default, Interface Builder uses the *Pane splitter*, a wide bar with a dimple in the middle. To keep up with Apple's trend-setting iTunes GUI, choose "Thin divider" instead. The graphical representation of the split view in the document window immediately changes to show that the divider is now a thin line.

Hold the pointer over the pop-up menu in the inspector to see a help tag describing the current selection. The help tag also identifies the Cocoa method you could use to get this attribute programmatically. In the case of split view dividers, it is the `-(NSSplitViewDividerStyle)dividerStyle` method. You know it is declared in `NSSplitView` because the Vertical Split View description in the Library window tells you so. You have to look up the applicable constants to use with this method, something you will learn how to do later. They are `NSSplitViewDividerStyleThick` and `NSSplitViewDividerStyleThin`. Interface Builder gives you a convenient GUI to set user interface values that you would otherwise have to set programmatically using Cocoa methods or functions. You will often find it useful to look up a method and its constants in the documentation to understand exactly what a particular Interface Builder setting does. These Interface Builder help tags make it easy to search for the applicable method documentation.

5. Position and resize the split view in the document window so that it fills the entire content of the window.

Selecting complex view objects in a window can be tricky. For example, if you click in the area occupied by the left or right custom view, you select the custom view instead of the split view holding both custom views. If you accidentally select one of the custom subviews and disturb its location within the split view, choose Edit > Undo, reselect the split view, and try again. To select and drag a container view like the split view, you can use any of several techniques. For example, drag a selection rectangle around the whole composite object until handles appear all around it. Or click one subview and then Option-click or Shift-click the other. Or double-click the Split View item in the document window in outline view mode. In the case of the Vertical Split View object, you can also click the vertical divider to select the entire object. By far the most useful technique, however, is to Shift-Control-click any object in the design surface.

This gesture brings up a contextual menu listing the entire hierarchy of objects under the pointer and allows you to select any one of them.

Once you have the split view selected, drag it up and to the right. As soon as you hold down the mouse button, the pointer changes to the open hand cursor, signaling that you can drag the view under it. Be careful not to drag the view into the toolbar or beyond the edges of the window. Interface Builder makes positioning it exactly in the upper-left corner of the content area easy by subtly snapping it to that position when it comes close, and dotted guide lines appear along the top and left edges of the window showing the destination of the snap. While you were dragging the split view toward the corner, you may have noticed other guides appear momentarily. Interface Builder provides those to help you position views in conformity to Apple's *Human Interface Guidelines*.

Resize the split view by dragging its lower-right handle into the lower-right corner of the window. The cursor changes to a small image indicating that you can drag the corner diagonally. The current dimensions of the view appear in a small overlay as you drag, for use when you want to size a view to specific dimensions.

Drag the divider to the left until the left pane is about a quarter of the width of the window. Interface Builder automatically resizes the custom subviews appropriately.

6. Finally, set the split view's autosizing behavior so that the split view continues to fill the window when the user resizes it.

With the split view selected in the document window, open the Split View Size inspector and look at its Autosizing section (**Figure 2.9**). The inner square in the diagram represents the frame of the selected view, and the outer square represents the frame of its containing superview. The lines connecting the two squares and the lines inside the inner square are referred to as *springs and struts*. In older versions of Interface Builder, springs were actually depicted as spring-like looping lines.

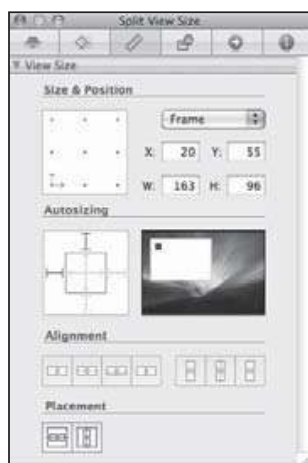


FIGURE 2.9 The Split View Size inspector.

Before proceeding, read the “Springs and Struts” sidebar for a complete description of Interface Builder’s autosizing capability.

Springs and Struts

It is easy to misunderstand how the springs and struts work. Get it right now, or you will lose a lot of time down the road struggling to make your windows work correctly when they are resized.

The lines in the margin between the inner and outer squares are rigid struts, as indicated by their squared-off end markers. Click a strut to toggle it between a solid line and a dotted line. When a strut is a solid line, it is enabled, and that margin remains fixed even as its superview resizes. When a strut is a dotted line, it is disabled, and the margin between the selected view and its superview changes without constraint as the superview resizes.

The lines within the inner square are flexible springs, as indicated by their arrow tips. Click a spring to toggle it between a solid line and dotted line. When a spring is a solid line, it is enabled, and the selected view resizes as the size of its superview changes. When a spring is a dotted line, it is disabled, and the selected view cannot change size as its superview resizes.

One internally inconsistent combination of springs and struts is possible—namely, to enable both struts in a given dimension, horizontal or vertical, while leaving the spring disabled in the same dimension. The selected view cannot resize because it has no spring, and so at least one margin must be allowed to vary as the superview resizes. Interface Builder resolves the inconsistency by applying a priority rule. Interface Builder anchors the selected view relative to the origin of the superview in its lower-left corner, ignoring the right strut or the top strut, depending on the dimension in which the inconsistency exists.

When you hold the mouse over the Autosizing section of the Size inspector, the graphic to the right of the diagram animates continuously to show the behavior of the current springs and struts settings in the diagram. This is an invaluable tool for understanding how your settings will work at run time.

Apple also provides sample code for the Sproing application, a developer utility that lets you experiment with the springs and struts settings in an animated graphic that is even more revealing than Interface Builder's own animated inspector. If you have trouble understanding springs and struts and making them do what you want, build the Sproing sample code project and keep the utility in your collection of developer utilities. It's a little old—in fact, it uses the old looped springs graphics—but it works as advertised.

(continues on next page)

Springs and Struts *(continued)*

Walk through the permutations in the horizontal dimension now, keeping the pointer in the Autosizing section of the inspector so that you can see how it works. It works the same in the vertical dimension from bottom to top.

- Disable the horizontal spring and both of the horizontal struts. The selected view (the inner square) remains fixed in width because its inner spring is disabled. The margins between the frame of the selected view and the frame of its superview (the outer square) vary freely on both sides with changes in the size of the superview because both struts are disabled.
- Enable the strut on the left margin, leaving the spring and the right strut disabled. The selected view remains fixed in width because it still has no spring, but now the left margin is fixed, too. The selected view is anchored in place relative to the left edge of the superview.
- Enable both the left strut and the right strut, leaving the spring disabled. The right strut is ignored because the selected view still has a fixed width due to the lack of a spring. This is the inconsistent state described earlier. It behaves like the preceding permutation, as if the right strut were still disabled.
- Disable the left strut, leaving the spring disabled but the right strut enabled. The selected view remains fixed in width because it still has no spring, but now the right margin is fixed. The selected view is anchored in place relative to the right edge of the superview.
- Enable the spring and both struts. Both margins remain fixed, and the selected view is anchored relative to both the left and right edges of the superview. The selected view resizes with the superview because, for the first time, the spring is enabled.
- Disable the right strut, leaving the left strut and the spring enabled. The selected view remains anchored relative to the left edge of the superview. The selected view resizes to the right, and at the same time the right margin varies, because the selected view's width is flexible and its right margin is not anchored relative to the right edge of the superview. The selected view and the right margin share their flexibility, with both changing proportionally as the superview resizes.
- Disable the left strut, too, leaving only the spring enabled. The selected view resizes both to the left and to the right, and at the same time the left and right margins both vary, because all three share their flexibility proportionally.

You never need to change the default autosizing settings of the window's Content View, because Cocoa automatically maintains the correct autosizing behavior for it.

Start by enabling the struts in all four margins of the Autosizing diagram for the Vertical Split View object, and by enabling the springs in both dimensions. To enable springs and struts, simply click them to make all of the dotted lines in the diagram solid. This has the effect of freezing the dimensions of the margins around all four edges of the selected view (the split view, represented by the inner square) and the corresponding edges of its containing superview (the window's content view, represented by the outer square), while leaving the split view free to resize in both dimensions. Since you have already set the edges of the split view to coincide with the edges of the document window and its content view, the split view completely fills the content view and its window while the user resizes the window.

Next, set the autosizing behavior of the left and right panes of the split view to make the left pane remain at a fixed width while the window resizes, as the left panes do in iTunes and the Finder. You will soon discover that this doesn't work without writing some code, but you should carry out this exercise anyway to understand how autosizing is done. You'll fix the problem shortly.

Select the custom view on the left. Then, in the View Size inspector, enable the top, left, and bottom struts and the vertical spring while disabling the horizontal spring and the right strut. The left pane is now anchored to the top and bottom edges of the containing split view while being flexible in height, so that it fills the split view vertically as the window is resized. At the same time, it is set so that it remains anchored to the left edge of the containing split view, and its width should remain fixed horizontally, while leaving the right margin flexible for the right pane—just the behavior you want.

Finally, set the autosizing behavior of the right pane of the split view. Select it and enable all of its springs and struts. The right pane is free to resize both horizontally and vertically, while all four of its edges remain anchored relative to the edges of the containing split view. Although its left edge should remain anchored relative to the left edge of the split view, it should remain separated from that edge by the width of the fixed-width left pane.

7. Now test the configuration of the split view.

Don't make the mistake of trying to test it by resizing the document window in Interface Builder. If you do, the window's size changes, but the sizes of the views within the window remain unchanged. Interface Builder is a design tool, and it thinks you want the window to be larger or smaller than the split view. If you hold down the Command key while you resize the window, all of its contained views resize with the window according to the springs and struts settings currently in effect, but Interface Builder still thinks you want to permanently change their size.

To test the split view's behavior without changing any settings, choose File > Simulate Interface to launch the Cocoa Simulator. The window appears in Cocoa Simulator exactly as it will appear in the finished Vermont Recipes application. Drag the window's resize control to make the window larger, and you see that the split view resizes to match.

Unfortunately, as predicted, the thin divider does not remain where it is to maintain the original fixed width of the left pane. This is intended. In most cases, a split view's panes should adjust proportionally as their container is resized. If you want to keep one of them fixed, you must write a little code.

8. Most views in the Cocoa AppKit come with built-in hooks that you can use to respond whenever the user makes changes to the view. With some of the hooks, Cocoa runs your code immediately before Cocoa draws the user's change to the screen, so that you can alter what Cocoa draws or take other steps in preparation for the change. With other hooks, Cocoa runs your code immediately after Cocoa draws the user's change. There are even hooks that let you prevent the user's change from happening at all.

These hooks come in several forms. Two of them are delegate methods and notifications. Cocoa posts a notification to a notification center where any object can register to observe it. A delegate method is more specific. Only one object can implement a particular delegate method, and if it does, only that one object receives the message when the user makes a change to a view. The receiving object must implement the delegate method, and it must be designated as the sending object's *delegate*. Notifications and delegate messages usually include information about the object that posted the notification or that sent the delegate message, enabling the observer or the delegate to take very sophisticated, context-aware action in response.

The `NSSplitView` class declares a delegate method known as `-splitView:resizeSubviewsWithOldSize:`, which is perfectly suited to the task of fixing the width of the left pane of the split view while the user resizes the window. This delegate method is called repeatedly as the user resizes the window, so the effect is smooth and continuous. It is available in Leopard and Snow Leopard, so you will implement it here. Shortly, you will implement a newer method that was introduced in Snow Leopard specifically to make it even easier to freeze a pane of a split view.

Your strategy in implementing the `-splitView:resizeSubviewsWithOldSize:` delegate method is to calculate the difference between the new width of the split view and its old width, and then to apply the entire difference to the right pane because the width of the left pane (and of the divider) is to remain unchanged.

If you confine your reading to Apple's *NSSplitView Class Reference* document, you might think the necessary information about the old width of the right

pane isn't available to the delegate method. The document's description of this delegate method says, "The size of the NSSplitView before the user resized it is indicated by *oldSize*," but it doesn't say where to find the old sizes of the two subviews within the split view. What to do?

Whenever you're unsure what a Cocoa method does and the documentation doesn't spell it out, look at the header file. In the Vermont Recipes project window, expand the Frameworks group and its Other Frameworks subgroup, and then expand the AppKit.framework and its Headers folder. You see a list of all of the AppKit's built-in framework headers. Scroll down to NSSplitView and double-click it to open the header file in a separate window. Press Command-F and search for *resizeSubviewsWithOldSize*. When you find the declaration, you see a comment that says, "Given that a split view has been resized but has not yet adjusted its subviews to accommodate the new size, and given the former size of the split view, adjust the subviews to accommodate the new size of the split view."

There's your answer: The split view that Cocoa passes into the delegate method in the `splitView` parameter has its new size already set, but the sizes of its subviews have not yet been adjusted. The name of the delegate method describes exactly this, indicating that the method starts with the old size of the split view but lets you resize its subviews. In other words, the subviews of the split view in the `splitView` parameter have not yet been adjusted and therefore still contain the old sizes. All you have to do is capture the old right pane from the `splitView`, add the difference between the new width and the old width of the enclosing split view to the right pane's old width, and set the right pane's frame to the new width, and you're good to go.

In Xcode, open the `RecipesWindowController.m` implementation file and add this method at the end, immediately before the `@end` directive:

```
- (void)splitView:(NSSplitView *)splitView
    resizeSubviewsWithOldSize:(NSSize)oldSize {
    NSView *rightPane = [[splitView subviews] objectAtIndex:1];
    NSRect rightFrame = [rightPane frame];
    rightFrame.size.width +=
        [splitView frame].size.width - oldSize.width;
    [rightPane setFrame:rightFrame];
    [splitView adjustSubviews];
}
```

Make sure you spell the name of the delegate method correctly, including correct capitalization. If you name it `-splitView:resizeSubViewsWithOldSize:`, for example, it won't work because Cocoa doesn't declare a delegate method with the *v* in *Subviews* capitalized. Cocoa would never call your misspelled delegate

method. This is a common error, and it is very hard to find unless you follow the first rule of prudent debugging: Check the spelling (including capitalization).

The last statement in the method calls `NSSplitView`'s `-adjustSubviews` method to fill the split view vertically as the user resizes it. It isn't necessary to assign specific values to the two subviews' frame height because `-adjustSubviews` does that for you. The only reason you had to set the subviews' widths to calculated values was because `-adjustSubviews` would have adjusted them proportionally. This isn't an issue in the vertical dimension, which is not split.

9. You must do one more thing to keep the width of the left pane of the split view constant. Designate the `RecipesWindowController` as the split view's delegate. It is very easy to forget to do this. Whenever your delegate methods don't work, the first thing to check (after the spelling) is whether the `delegate` outlet was connected. In Interface Builder, select the split view, open the Split View Connections inspector, and drag from the open circle adjacent to the `delegate` outlet to the File's Owner proxy in the nib file's document window. Then save the nib file.
10. Snow Leopard introduces a new `NSSplitView` delegate method, `-splitView:shouldAdjustSizeOfSubview:`, specifically to make it easier for you to freeze a pane of a split view while its window is being resized. It is intended to save you the trouble of figuring out how to implement the `-splitView:resizeSubviewsWithOldSize:` delegate method. You have already implemented the older delegate method, which works in Leopard and still works in Snow Leopard, so you don't have to implement the new delegate method in Vermont Recipes. Nevertheless, this book is focused on Snow Leopard, so you'll implement the new delegate method as well.

By convention, any delegate method that includes the term *should* returns a Boolean value of YES or NO. The new Snow Leopard delegate method, `-splitView:shouldAdjustSizeOfSubview:`, conforms to this convention. In order to freeze one of the split view's subviews, you implement the delegate method so that it returns NO when its second parameter contains the subview that should be frozen. A return value of NO from a *should* delegate method causes Cocoa to refuse to execute the action. A return value of YES allows the action to be executed.

Add this delegate method to the `RecipesWindowController.m` implementation file:

```
- (BOOL)splitView:(NSSplitView *)splitView
    shouldAdjustSizeOfSubview:(NSView *)view {
    if ([splitView isVertical] &&
        (view == [[splitView subviews] objectAtIndex:0])) {
        return NO;
    }
    return YES;
}
```

The `if` test is written to distinguish between the vertical split view you intend to address with this delegate method and the horizontal split view, which you want to ignore. Since both kinds of split view will exist in the recipes window, this delegate method will be called for both of them while the user is resizing the window. By testing whether the `splitView` argument in any invocation of the method is a vertical split view, you avoid returning `NO` for the horizontal split view. You return `YES` for it, instead, allowing both panes of the horizontal split view to resize proportionally, as all split view panes do by default.

If this invocation is for the vertical split view, you also test to see whether the subview for which it is being invoked has an index of `0` in the array of subviews. If so, this invocation is for the left pane, so you return `NO` to prevent it from resizing. If it is the right pane, execution falls through the `if` clause and the delegate method returns `YES`, allowing the right pane to resize.

These tests may not work appropriately if you later add another vertical split view to the recipes window—say, in one of the tab views you will add shortly. If you don't want to freeze the pane having index `0` in that vertical split view, you would have to devise an additional test to distinguish between the two vertical split views. The most general way to do this would be to add an outlet for the split view you intend to address, connect it in Interface Builder, and in this delegate method test whether the `splitView` is equal to the outlet. If you're a defensive programmer, that's the way you would do it, but live dangerously for now and simply make a note to yourself to fix this if you do add another vertical split view later.

11. There is one problem you should fix now. You have implemented two delegate methods that accomplish the same goal. This is inefficient, and worse, there is a theoretical risk that it might cause errors at run time. It won't be a problem when Vermont Recipes is running under Leopard, because the `-splitView:shouldAdjustSizeOfSubview:` delegate method isn't declared in Leopard. It was introduced in Snow Leopard, so it simply won't be called under Leopard. However, when Vermont Recipes is running under Snow Leopard, both methods will be called.

To fix this problem, you need to insert a statement at the beginning of the `-splitView:resizeSubviewsWithOldSize:` method to detect whether it is running under Snow Leopard. If so, prevent the duplicate code from executing. Revise the method like this:

```
- (void)splitView:(NSSplitView *)splitView
    resizeSubviewsWithOldSize:(NSSize)oldSize {
    if (floor(NSAppKitVersionNumber)
        <= NSAppKitVersionNumber10_5) {
        NSView *rightPane = [[splitView subviews] objectAtIndex:1];
```

```

        NSRect rightFrame = [rightPane frame];
        rightFrame.size.width +=
            [splitView frame].size.width - oldSize.width;
        [rightPane setFrame:rightFrame];
    }
    [splitView adjustSubviews];
}

```

The test you add at the beginning of the method is based on a Cocoa versioning mechanism that has been in use throughout the lifetime of Mac OS X. AppKit comes with a built-in version constant, `NSAppKitVersionNumber`, which is revised with each major revision of Mac OS X. AppKit also defines the AppKit version number that belongs to each major revision of Mac OS X, such as `NSAppKitVersionNumber10_5` for Mac OS X 10.5 Leopard. You use the C standard library function `floor()` to get the largest integer that is not less than the version of AppKit on the currently running computer, then compare it with the version constant for the version of Mac OS X for which you are testing. You need to use `floor()` to get the integer part of the version number, because the fractional part is sometimes used to identify minor revisions of AppKit between major versions of Mac OS X. These AppKit version numbers and their usage are documented in Apple's *Cross-Development Programming Guide*, and the AppKit Release Notes for every new major version of Mac OS X have a review of them near the beginning.

Here, you test whether the version of AppKit installed on the running computer belongs to Leopard or older. Since Vermont Recipes will not run on Mac OS X 10.4 Tiger or older, you are effectively testing whether the computer is running Leopard. If so, you execute the body of the `-splitview:resizeSubviewsWithOldSize:` method to freeze the left pane, because the newer delegate method won't run on Leopard. If the computer is running Mac OS X 10.6 Snow Leopard, the test is evaluated as `NO` and the body of the method is not executed.

The last line—the statement `[splitView adjustSubviews];`—is run in both cases because it lies outside the `if` block. It is needed under Leopard, as explained earlier, because it adjusts the non-split dimension of the split view. It is needed for that reason under Snow Leopard as well, but also for another reason. The new Snow Leopard delegate method, `-splitView:shouldAdjustSizeOfSubview:`, is designed to modify how the `-adjustSubviews` method works, so the delegate method is called only if you call `-adjustSubviews`. Here, Cocoa calls the `-splitView:resizeSubviewsWithOldSize:` method whenever the user resizes the window, so it is a perfect place to call `-adjustSubviews` even under Snow Leopard. If you were writing Vermont Recipes for Snow Leopard only and did not care about Leopard functionality, you would instead implement the `-splitViewWillResizeSubviews:` delegate method to call `-adjustSubviews`, and `-splitView:shouldAdjustSizeOfSubview:` would be executed just the same.

12. Test your work. You can't test the autosizing behavior of the left pane by choosing File > Simulate Interface, because the Cocoa Simulator does not take into account code you have written in your custom classes. It works only with settings within Interface Builder itself. Try it. When you resize the window, the left and right panes resize proportionally.

Instead, build and run the application in Xcode. When the application launches and the window opens, resize the window. You finally see that the left pane remains fixed in width as you resize the window. You have achieved your goal of emulating the behavior of iTunes.

Step 4: Add a Horizontal Split View

The iTunes-like single-view user interface that has become so popular has a horizontal split view in the right pane of the vertical split view. This allows you to implement what I've been calling a master-master-detail view arrangement in which the user makes basic category choices in the left pane, then makes subchoices in the upper pane on the right, and then finally sees the details of this dual selection in the lower pane on the right.

Leave the content of the top pane of the horizontal split view undefined for now, just as the left pane of the vertical split view is undefined.

The process is identical to what you did in Step 3 to create the vertical split view, except that for now you'll allow both panes to resize proportionally when the user resizes the window.

1. Use the pop-up menu at the top of Interface Builder's Library window to choose Library > Cocoa > Views & Cells > Layout Views.
2. Drag a Horizontal Split View object out of the Library window and drop it anywhere in the right pane of the vertical split view in the recipes window.
3. Leave the Style setting in the Split View Attributes inspector set to "Pane splitter." Apple's iTunes, Mail, and Xcode applications do this as well, because it gives the user a means to see the divider and drag it even when one of the panes is fully collapsed.
4. Position and resize the horizontal split view to fill the right pane of the vertical split view.
5. Set the springs and struts of the horizontal split view in the Split View Size inspector so that all of them are enabled. This will force the split view to resize so that it always fills the entire right pane of the vertical split view. All of the subviews' springs and struts should also be enabled, for the same reason.

Step 5: Add a Tab View

Now add yet another view to the recipes window, this time a tab view in the lower pane of the horizontal split view. Every food recipe in Vermont Recipes 2 includes several sections, such as ingredients and instructions. Rather than try to cram them into one window all at once, you can reduce clutter and allow room for future enhancements by letting the user switch between tab views to limit visible information to whatever is relevant to the task at hand.

You should have the drill down pat by now.

1. Use the Library window's pop-up menu to return to the list of views at Library > Cocoa > Views & Cells > Layout Views.
2. Drag the Tab View object and drop it anywhere in the bottom pane of the horizontal split view in the right pane of the vertical split view. The image expands into a full-fledged tab view with two tabs at the top.
3. Examine the view hierarchy in the nib file's document window in outline view mode. When you fully expand the Recipes Window entry, you see that a Top Tab View object now resides in the second, or bottom, Custom View of the horizontal Split View, which in turn resides in the second, or right, Custom View of the vertical Split View. You also see that each Tab View Item in the Top Tab View—both the selected tab view item and the other tab view item—contains a View object.
4. Reposition and resize the tab view so that it completely fills the bottom pane of the horizontal split view. The side and bottom edges should butt up against the edges of the bottom pane, but you should let the top of the tabs snap to the dotted horizontal grid line that appears a short distance below the top of the pane. You will review Apple's *Human Interface Guidelines* with respect to tab view placement later, but for now you want to keep visual clutter to a minimum because this is already a complex window (**Figure 2.10**).



FIGURE 2.10 The document window after adding a tab view.

5. Set the Tab View's springs and struts so that the tab view resizes with the window while its edges remain anchored to the frame of the enclosing right split view. In the Tab View Size inspector, enable all four struts and both springs.

6. It is surprisingly easy to forget to set a new view's or control's autosizing behavior. As a preventive measure, get in the habit of resizing a window using the Cocoa Simulator every time you add a view or control to the window, not only to make sure it is working correctly but also to see what you've forgotten.

Choose File > Simulate Interface, and resize the window. The tab view continuously resizes in both the horizontal and vertical dimensions to fill the bottom pane of the horizontal split view, as expected.

Don't be alarmed when you see that the left pane of the split view now changes width proportionally instead of remaining fixed. You learned in Step 3 that the Cocoa Simulator does not exercise the delegate method you added to the window controller. To see the width of the left pane remain constant while the right pane and its tab view change size, you must build and run the application in Xcode.

Step 6: Add a Drawer

One of the features called for by the Vermont Recipes application specification is to identify the sources of recipes contained in the database. This information is not needed while preparing a meal, so it should be out of view most of the time. It should nevertheless be easily accessible while the user is browsing the database for ideas or information. It could be placed in a tab of the tab view you just added, but for consistency's sake you should plan to keep the tabs of the tab view focused on food preparation. The user might want to view the source of the recipe while looking at any of the tab views.

A perfect user interface element for this purpose is a drawer. A drawer is a separate window that slides out from behind the main window when needed, just as a drawer slides out of a desk or cupboard. The idea is that a drawer should contain information that is related to the information in the window, but which is not central to the function of the window and need not be visible all the time.

In Mac OS X, a drawer slides out from either side of the main window or from its top or bottom. It remains attached to its parent window at all times, whether the parent window is moved, resized, minimized to the Dock, or hidden. It has no title bar and no close, zoom or minimize buttons of its own. Typically, it slides in or out when the user clicks a button in the parent window dedicated to that purpose.

Create the drawer now without contents, leaving its subviews and controls later.

1. First, give the existing document window object a distinctive name. Select the Window object in the document window, select its label for editing, and change it from Window to **Recipes Window**.

The new label only affects the window object, not the title of the window in the application's user interface. You could change the window's title in its title bar as well, by selecting the newly renamed Recipes Window object, selecting the Window Attributes inspector, and entering **Vermont Recipes** in the Title field. If you did this, you would see the title of the document window change to Vermont Recipes in Interface Builder. Don't do this, however, because a document-based application controls its document window's titles in code, ignoring the title you give the window in Interface Builder. It does this because the window's title bar should display whatever name the user gives the document. Each document's window may therefore have a different title, or Untitled if the user hasn't yet saved it.

2. Now you're ready to add a drawer. Start by returning to Interface Builder's Library window. In older versions of Interface Builder, the Library contained a separate drawer object that you could drag into the design surface. You won't find a separate drawer object in the Objects tab view of Interface Builder 3.2, although the combination of a window, a drawer, and the drawer's content view is available in the pop-up menu at Library > Cocoa > Application > Windows > Window and Drawer. You want to attach a drawer to the document window you have already created, not to create a new window.

One possibility is to switch to the Classes tab view in the Library window and locate the `NSDrawer` class. You can drag any class from the Classes list into the nib file's window to instantiate an object of that class. If you do it this way, however, you'll also have to drag an `NSView` class to hold the drawer's content, and you'll have to figure out how to connect everything to the existing window. If you already know what is required, this may in fact be the easiest way to do it.

You don't yet know what to connect, however, so go ahead and use the combined object. You'll be able to examine the connections that come with it. With the Objects tab view selected, drag a Window and Drawer object to the document window. As you drag it, you see the combined window and drawer image in the Library window separate into three distinct objects, a new Window, a new Drawer Content View, and a new Drawer, all of which end up in the document window (**Figure 2.11**). Your strategy is to examine the connections of each, to redirect them to your existing split view window as appropriate, and then to discard the new window that came across with the drawer and its content view.



FIGURE 2.11 The RecipesWindow.xib window after adding a Window and Drawer object.

3. You are focused on hooking up the Drawer object, so examine its features before turning to the new Drawer Content View and parent Window objects. Select the Drawer object in the document window, and then select the Drawer Identity inspector. You see that it is an object instantiated from the `NSDrawer` class. In fact, it is exactly what you would have found here if you had dragged an `NSDrawer` object from the Classes tab view of the Library window, except that it now has some connections, as you will see in a moment.

Next, select the Drawer Attributes inspector. It has only a single attribute, the edge of the parent window to which it is to be attached. It is already set to the right edge, which is what you want.

Then select the Drawer Size inspector. It doesn't contain the usual springs and struts control or other size settings. Instead, it contains several new settings reflecting the fact that a drawer cannot be sized independently but only in relation to its parent window. For the time being, accept the default values. They work fine as is for purposes of the simulator, and you can change them as needed later, when you start adding content to the drawer.

Finally, select the Drawer Connections inspector. You see that the `contentView` and `parentWindow` outlets are already connected. These two outlets represent instance variables of the same name that are declared in Cocoa's `NSDrawer` class. Because Interface Builder has connected them to existing objects in the nib file, you will not have to write code to assign values to them. You will be able to refer to them in your code, and they will already have the values you set up in the nib file.

The `contentView` outlet is connected to the new Drawer Content View object, which is just what you want. However, if you hold the pointer over the `parentWindow` outlet, you see the new Window object highlight. You want your existing split view window, now named Recipes Window, to be the drawer's parent window. To change the connection, click the Clear (x) button adjacent to the reference to the Window object to break the connection, and then click the now-empty circle to the right of the `parentWindow` outlet and drag to the recipes window. You did not have to disconnect the outlet first; you could simply have connected the `parentWindow` outlet to the split view window to replace the previous connection. Now when you make the Window Connections inspector active and hold the mouse over the `parentWindow` outlet, the existing window for the Recipes Window object highlights.

4. Select the Drawer Content View object in the document window. If you select the View Identity inspector, the View Attributes inspector, and the View Size inspector in turn, you see that it is an ordinary `NSView` object in every respect.

Select the View Connections inspector. You see that it has a referencing outlet named `contentView` that is connected from the Drawer object. This is the same connection between the Drawer and the Drawer Content View object that you saw a moment ago when you were examining the Drawer object, but you're now looking at it from the other end. That's the difference between an outlet, which is an outgoing reference, so to speak, and a referencing outlet, which is an incoming reference. When you make the View Connections inspector active and hold the pointer over this connection, the Drawer object's icon in the document window highlights. This is exactly what you want, so you don't have to change anything here.

5. Select the new Window object (not the newly renamed Recipes Window object) in the document window. Even if you hadn't given either window a title, you could tell them apart by double-clicking them. Double-click the new Window object now, and it opens as an empty window. This is the window object that you created when you dragged the Window and Drawer item from the Library window. Its icon is named Window (Window). Be sure it is selected in the document window, and then select the Window Connections inspector. You see that this window has no connections. When you first created it a moment ago, it had a referencing outlet from the Drawer object named `parentWindow`, but you changed that to point to the Recipes Window object.

Since you no longer need the new Window object, and its connections have been removed, select it in the document window (make sure you select the right one), and press the Delete key.

6. To make sure everything is going as expected, select the Recipes Window object in the document window and select its Window Connections inspector. You see the `delegate` outlet and the `window` referencing outlet, which you remember from Step 1. You also see the other end of the `parentWindow` referencing outlet from the Drawer, which you created a moment ago. Everything looks right.

Step 7: Add a Toolbar Item to Open and Close the Drawer

You're ready to create your first real control and an associated action, now that the stage is set with several views. Begin by adding a toolbar item to the right end of the toolbar to open and close your new drawer.

1. In the document window, double-click the toolbar. A sheet opens below it labeled Allowed Toolbar Items (**Figure 2.12**). The sheet contains all of the toolbar items, or buttons, that a user can install when customizing the toolbar. The toolbar itself represents the default set, which need not contain all of the allowed items. You need to add a toolbar item to the allowed set, and it should also be included in the default set so that the user can always open and close the drawer.



FIGURE 2.12 The Allowed Toolbar Items sheet.

2. To make room for the new toolbar item, remove the Customize toolbar item from the default set. Drag it from the toolbar to the desktop to delete it. Be careful not to drag it from the Allowed Toolbar Items sheet, because you want to keep it available in case the user likes it in the toolbar.

Customizing a window's toolbar is not such a regular user activity that it needs its own button in the default toolbar. The user can always customize the toolbar by choosing the View menu's standard Customize Toolbar menu item. If any user wants it in the toolbar, it remains available in the allowed set.

3. Select the Media tab in the Library window and scroll down until you find the NSInfo image, consisting of the lowercase letter *i* in a blue circle. Snow Leopard comes with a large number of images available for use in your applications.
4. Drag the NSInfo image from the window and drop it into the Allowed Toolbar Items sheet to the right of the Customize toolbar item. It is now an allowed toolbar item for this window.
5. Double-click the new toolbar image's label, NSInfo, and change it to **Recipe Info**.
6. You've created your first control; now you're ready to implement your first action. Control-click (right-click) the Drawer object in the document window. A Heads Up Device (HUD) window opens, listing, among other things, three available received actions, "close:," "open:," and "toggle:." These correspond to three action methods declared in Cocoa's NSDrawer class, `-close:`, `-open:`, and `-toggle:`. You will learn more about action methods later, but for now you only need to know that an action message, when received by an object of the class that declares a method of the same name, causes the method to execute. You want your new

Recipe Info toolbar item to open the drawer when it's closed and to close the drawer when it's open, so it seems that the **toggle:** action is the one to use.

Pause for a moment to verify this in the documentation. A good place to look first for the appropriate action method is in *NSDrawer Class Reference*. In Xcode, choose Help > Documentation. In the Documentation window, search for *NSDrawer* and open the *NSDrawer Class Reference*. All of Apple's Class Reference documents have a section near the beginning named "Tasks," and *NSDrawer*'s "Tasks" section includes a topic called "Opening and Closing Drawers." There you find a description of the **-toggle:** method and, indeed, it seems suitable.

Now make the connection. Drag from the empty circle beside the **toggle:** action in the HUD to the *NSInfo* image, now named Recipe Info, in the Allowed Toolbar Items sheet. The HUD shows that the Recipe Info toolbar item has been connected to the **toggle:** action in the Drawer object. When the user clicks the Recipe Info button, it will send the **toggle:** message to its target, the drawer, causing the drawer's **-toggle:** method to be executed. The drawer opens or closes, depending on its state at the time the message was sent.

7. Drag the new toolbar image from the Allowed Toolbar Items sheet and drop it at the right end of the toolbar. You may have to close the sheet first by clicking Done, and then reopen it by clicking the toolbar. The Recipe Info button is now in the default set for this window's toolbar. Click Done to close the sheet.
8. Now you can test your new drawer in the Cocoa Simulator. Choose File > Simulate Interface. The document window looks identical to the window that will eventually appear in the completed Vermont Recipes application. You see the Recipe Info toolbar item at the right end of the toolbar. Click it, and your new drawer slides out to the right. Click it again, and the drawer slides closed.

Go wild. Open the drawer again and drag its outside edge to the right to make it wider. Close it and reopen it, and you see that it automatically remembers its last size. Click the window's zoom button while the drawer is open, and the window jumps to full size, filling the screen except for the room it leaves for the drawer that is still sticking out from the window's right edge. Click the zoom button again, and then click the minimize button. The window returns to its original size, and then it minimizes to the Dock. Double-click its icon in the Dock, and it returns to full size with the drawer still visible. Finally, use the window's resize control to change the window's size continuously. The drawer remains open no matter how wide, tall, narrow, or short you make the window. Click the Recipe Info toolbar item again, and the drawer slides closed.

For now, don't worry about the empty view you might have noticed floating in space when you run the Cocoa Simulator. Once you start filling the Drawer Content View object with views and controls, the content view and its contents will appear in the drawer, as they should.

The Target-Action Design Pattern

The technique you used in Step 7 is known as the *Target-Action* design pattern. Like MVC, it plays a central role in the design of Cocoa applications. The concept is really very simple. An object sends a message to a target, and the target responds.

You read something about this already in the “Outlets and Actions” sidebar at the beginning of this recipe. In Cocoa, the sender of an action message may be any kind of object, but it is easiest to understand if you think of it as a menu item or control. When the user chooses a menu item or, say, clicks a button, the menu item or control sends an action message that it has been programmed to know how to send. It typically sends the action message to a specific target, which it has also been programmed to know about. This works only if the specified target object knows how to respond to this particular action message, and that’s where you come in. You write an Objective-C method in the target class that knows how to react usefully when it receives the action message. You can do all this in Interface Builder, for example, by drawing connections from a sending object to a target object and then choosing an action. Or you can do it in code.

You might more sensibly speak of a *target-action-sender* design pattern, since there are really three parts to it, but we’re stuck with the jargon we have. The three-part concept is embodied very concretely in the Objective-C methods you write to implement this design pattern. In Objective-C, we speak of a *receiver*, which is an object that receives messages, and we speak of a *message*, which in more conventional terminology might be called a *method call* or *procedure call*. The *receiver* is the *target* of the *message*, and to make this work, the *receiver* declares a *method* that responds to the *message*. The method and the message have the same name, or signature, because one is the method and the other is simply a method call. In Objective-C’s traditional bracketed notation, you send a *message* to its *target*, or *receiver*, like this: `[receiver message]` or `[receiver message:parameter value]`. In the case of an action message, you always use the latter form, in which the message takes a single parameter, referred to as the *sender*, like this: `[target action:sender]`. Because of the *sender* parameter, the target always knows where the message came from.

In Cocoa, important functionality is based on action messages that *don’t have a target*. How can this be? Under the hood, it means that the target, or receiver, is *nil*. You can’t send a message to *nil* in Objective-C, of course—that is, doing so doesn’t cause an error, but it doesn’t do anything, either. But Cocoa provides a mechanism called the *responder chain*, whereby a message that gets sent without a specific target is automatically sent to an object called the *first responder*. The first responder is not known at compile time. Instead, it is represented by a peculiar icon you have already seen in the document windows you’ve been working with, known as the First Responder proxy. You will learn the details later. For now, all you need to know is that Cocoa sends the message from object to object in the application’s user interface, starting with the view or control that currently has keyboard focus and proceeding according to a carefully defined path, until it finds an object that knows how to respond. That object is made the target of the action at run time. This mechanism makes it possible to write very dynamic and powerful code without doing much work.

Step 8: Build and Run the Application

Build and run the application just as you did near the end of Recipe 1. This should be your habit after you complete every recipe and, in your real life as a Cocoa developer, after every significant change to your projects. You will often discover that you've made a mistake and something doesn't work right. It is far better to discover that now, so you can find the problem and fix it while the project hasn't changed much since the last time you built and ran it. There will be much less new material for you to review.

Click the Build and Run icon in the Xcode project window's toolbar. Click Save All if you're told there are unsaved changes, and wait a few moments.

When the recipes window opens (**Figure 2.13**) and its menu bar appears at the top of the screen, check whether all your changes work. Resize the window to make sure the split view panes work correctly, with a fixed-width left pane, and that the tab view resizes as it did in the Cocoa Simulator. Click the Recipe Info item in the toolbar a couple of times to make sure the drawer opens and closes.

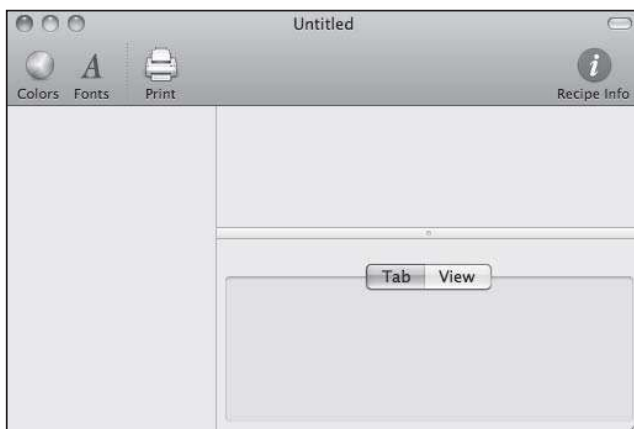


FIGURE 2.13 The window as it appears in Recipe 2.

Also check out the menu bar. Many of the menu items work, including File > New, which opens a second document window.

Step 9: Save and Archive the Project

Good housekeeping requires that you save and archive the project after every significant change. Quit the running application, close the Xcode project window, and save if asked to do so. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 2.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 3.

Conclusion

This is a good place to stop for now. You have learned a lot about Interface Builder and how to use it to set the stage—the graphical user interface—for the real work your application does. As you continue through the book, you will add more controls, menus and menu items, and views and windows, and you will learn how to store and retrieve data and use other features of Cocoa to create a complete and useful application.

Documentation

Interface Builder

As this recipe makes clear, Interface Builder is primarily a graphical user interface design tool. Apple considers consistency of design from application to application to be a hallmark of the Macintosh user experience. It will come as no surprise to you, therefore, to learn that Interface Builder includes features designed to make it easy to conform to the *Apple Human Interface Guidelines*. The *HIG*, as it is popularly known, is even linked in Interface Builder’s Help menu to deprive you of any excuse for failing to comply.

For detailed technical instruction on the use of Interface Builder’s features, read the *Interface Builder User Guide*. It, too, is linked in Interface Builder’s Help menu, as *Interface Builder Help*. Much shorter introductions are contained in the “Nib Files” section of the *Resource Programming Guide* and the “Interface Builder” section of Appendix C of *Mac OS X Technology Overview*.

As always, consult the release notes and other material for information about the latest release of Interface Builder. These include a long series of release notes, *Interface Builder 3.0 Release Notes*, *Interface Builder 3.1 Release Notes*, and *Interface Builder 3.1.1 Release Notes*. As of this writing, there are no Interface Builder 3.2 Release Notes, but watch for them.

There are a number of more technical or specialized documents, including the “Preparing Your Nib Files for Localization” section of the *Internationalization Programming Topics* document. You can even create an Interface Builder interface for your own custom controls, as explained in the *Interface Builder Plug-In Programming Guide*.

Create a Simple Text Document

In this recipe and several following recipes, you start to write some serious code. In the process, you learn more about how to use the Xcode code editor, in which you will spend most of your time as a Cocoa developer, as well as learning more about Interface Builder. You also learn common techniques for structuring your code files to make them easier to read and maintain over time.

You will create an auxiliary document with relatively simple data storage requirements to serve as the vehicle for these lessons. You were warned at the outset that, because Core Data is an advanced topic, you would shortly turn to simpler tasks to prepare you for an eventual return to Core Data. The techniques you learn in this recipe can be used in any application to set up an auxiliary document with its own window. As a bonus, the auxiliary document's window contains a text view that you will use to learn how to create a text editor with powerful editing and formatting features.

In this recipe you learn how to take advantage of the Cocoa text system to create a Rich Text Format (RTF) document, how to set up a split view to read and edit different parts of a text document in separate panes, and how to write the auxiliary document's data to disk and to read it back from disk. In subsequent recipes, you will turn to new lessons about adding controls and the wiring required to make them work, about configuring the application's main menu, and about polishing up the user interface. Along the way, you learn a little about a fundamental feature of almost any Mac OS X application, undo and redo.

HIGHLIGHTS

- Creating an auxiliary document*
- Using Xcode snapshots*
- Creating a new class in Xcode*
- Creating a new class in Interface Builder*
- Allocating and initializing objects*
- Adding and configuring a Cocoa text view*
- Subclassing NSDocumentController*
- Using Uniform Type Identifiers to identify document types and ownership*
- Writing data to disk and reading it from disk*

First, supplement the original Vermont Recipes application specification by adding a specification for the new, simple document you create in this recipe. It implements a unique but useful feature for a recipe application: a free-form diary in which the chef can record the experiences of an ongoing culinary life. In the application's user interface, you call the new document *Chef's Diary*. You name its class *DiaryDocument* for purposes of development. The Chef's Diary is a place where anybody, from a backyard cook just learning how to barbecue to the chef de cuisine at a five-star restaurant, can jot down informal notes on the spur of the moment without having to worry about organizing and categorizing them. Like any diary, it is organized chronologically, and each entry's title is the date and time of creation. To help the user find information in the diary later, its contents are searchable. In addition, it allows the user to add tags to any entry, in order to support tag-based search. The document stores text, and the text includes the dates and tags related to individual entries. This is an RTF document, so it supports extensive formatting features. Once saved, it can be opened and edited in any application that supports RTF, such as TextEdit, Pages, and Microsoft Word.

The document's data consists of free-form RTF text, and a prolific chef might write an unlimited amount of it. The standard Macintosh scrolling text view should therefore be the primary user interface element in the diary window. To allow the user to look at previous entries while typing a new entry, make two scrolling text views, one in each pane of a horizontal split view with a movable divider.

This will be a surprisingly full-featured word processor, but you don't have to do anything special to create most of its options. They come ready-made in Interface Builder. You will nevertheless place a few controls across the bottom of the window in Recipe 4. The specification calls for dated titles and searchable tags, so buttons to add them will be convenient: an Add Entry button to insert the current date and time as formatted text starting a new entry, and an Add Tag button to insert one or more tags following an entry's title. Buttons to jump to the previous and next entries as well as the first and last entries will also be useful. Also, supplement the scroll bar and the navigation buttons by including a control to jump to a particular entry by date. Finally, include a search field to find diary entries by tag.

Before getting to work, refine these ideas a little. Since a diary is chronological, it makes sense to add new entries at the bottom, immediately following whatever was last typed. A nice way to let the user jump to other entries by date would be to include a fully interactive date picker. This control should always display the date and time of the entry that is currently scrolled into view, updating automatically as the user navigates through the diary. Changing the date and time of the control should automatically scroll the text view to the corresponding entry and select its title. What if there is no entry for the date and time the user enters? Simply display

the oldest entry coming after the target date, or the newest entry in the document if there is none later than the target date.

To implement the Chef's Diary, you must create the `DiaryDocument` class and a window controller class you name *DiaryWindowController*. In addition, you must create another nib file, which you name *DiaryWindow*, and in it design a window containing views and controls to display and edit the diary. Finally, you must implement a means to store and retrieve the data controlled by the document.

You start by setting up the infrastructure in this recipe, including creating the code files and the nib file. You also add the text view and all the code needed to make it work. You will add controls to the document window and make them work in Recipe 4. In Recipe 5, you will configure the menu bar. Finally, in Recipe 6, you will make sure there can be only one Chef's Diary. In doing all of this, you will learn something about several basic Cocoa techniques, such as reference-counted memory management and validating controls and menu items.

Step 1: Create the DiaryDocument Class in Xcode

You learned how to use Xcode to create a new class in Step 7 of Recipe 1. Follow those same steps now to create the new `DiaryDocument` class. `DiaryDocument` is a subclass of Cocoa's `NSDocument` class, not of `NSPersistentDocument` as `RecipesDocument` is. This is because `DiaryDocument` will not rely on Core Data to manage its data. Since `NSPersistentDocument` is itself a subclass of `NSDocument`, `DiaryDocument` and `RecipesDocument` fill similar roles and respond to many of the same methods that both of them inherit from `NSDocument`.

Recall that `NSDocument` and its subclasses play the role of a specialized controller in the MVC design pattern—specifically, `NSDocument` is a *model-controller*. It works in tandem with `NSWindowController` to mediate two-way communications between the model that holds the document's data and the window and views where the user reads and edits the data. After you create `DiaryDocument`, you create the `DiaryWindowController` class and the *DiaryWindow* nib file with a window in which to display and manipulate the diary's data. `DiaryWindowController` is the diary document's *view-controller*.

1. Start by opening the Vermont Recipes 2.0.0 folder in which you saved the project folder at the end of Recipe 2. Leave the compressed project folder you archived at that time where it is, and open the working Vermont Recipes subfolder. In it,

double-click the Vermont Recipes.xcodeproj file to launch Xcode and open the project window.

Once again, you have a housekeeping matter to take care of first, incrementing the CFBundleVersion value in the Info.plist file. You could do this by opening the Vermont_Recipes-Info.plist file in the Resources group, but there is a slightly easier way. Select the Vermont Recipes target in the Targets group and click the Info button in the toolbar. Even easier, simply double-click the target of interest, the Vermont Recipes target. Then select the Properties tab and change the value in the Version field from 2 to 3, and then close the Target Info window. When you open the About window after building and running the application at the end of this recipe, you will see the application's version displayed as 2.0.0 (3).

2. In Xcode, choose File > New File to select a template.
3. Click Cocoa Class in the left pane of the New File window; then click the “Objective-C class” template's icon in the upper-right pane. In the lower-right pane, use the “Subclass of” pop-up menu to create a subclass of NSDocument.
4. Click the Next button. In the next window, enter **DiaryDocument** in the File Name field for the implementation file so as to name it *DiaryDocument.m*. Leave the checkbox to create the DiaryDocument.h header file selected.
5. Click Finish to create the new files in the Vermont Recipes project and its Vermont Recipes target. If the two new files aren't located in the Classes group in the Groups & Files pane, drag them there and drop them below the Recipes-Document files.
6. Most developers find it convenient to create subgroups within the Classes group as soon as a project starts to get complicated. In a multidocument application project, I normally create a subgroup named Documents and another subgroup named Window Controllers.

To follow this practice yourself, select the Classes group and choose Project > New Group. A group named New Group appears at the top of the Classes group, and its name is selected for editing. Type **Documents** and press the Enter key to commit the new name. Then select all four of the document code files and drag them onto the new Documents group. They now appear indented inside that subgroup, and you can collapse the Documents group to hide them when you aren't working on them.

7. Repeat the process to create a new Window Controllers subgroup. Position it just below the new Documents group, and drag the two window controller code files into it (**Figure 3.1**). In Step 3, you will create two new DiaryWindow-Controller code files and add them to the Window Controllers subgroup.

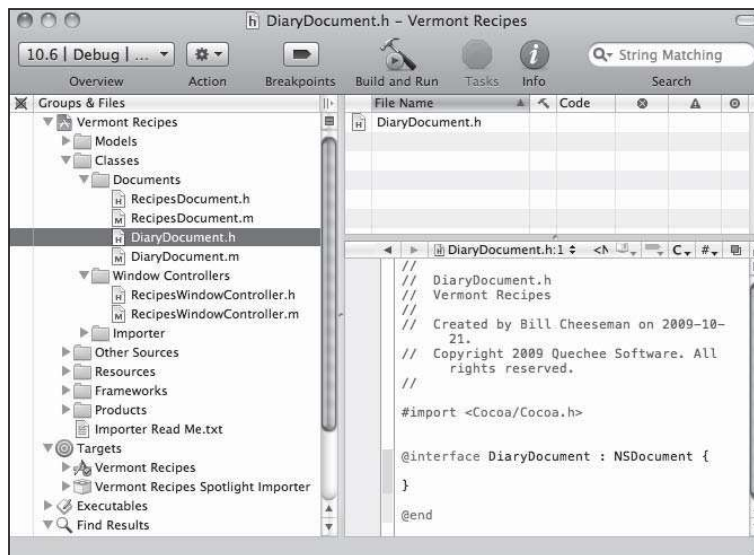


FIGURE 3.1 The Classes group with new subgroups.

8. Expand the Documents group, open the DiaryDocument header and implementation files, and edit the identifying information at the top of each of them as you did in Step 4 of Recipe 1.
9. The template inserted three method implementations into the DiaryDocument.m implementation file. You will use two of them, `-dataOfType:error:` and `-readFromData:ofType:error:`, in Step 7 to store the RTF text in the diary document to disk and to retrieve it from disk.

The third template method, `-windowNibName`, is familiar to you from Step 4 of Recipe 1, where you changed it to return `@“RecipesWindow”` instead of `@“MyDocument”` in the RecipesDocument class. You learned in Step 6 of Recipe 1 that this method can be used only in a document having a single kind of window, where `NSWindowController` can do its work without subclassing. You deleted `-windowNibName` from RecipesDocument because that document is designed to use more than one kind of window to display its contents, requiring you to create multiple subclasses of `NSWindowController` and to implement `-makeWindowControllers` to instantiate them. You should now delete the `-windowNibName` method from DiaryDocument too, not because DiaryDocument uses more than one kind of window, but because you need to subclass its window controller to customize its methods. In any application of even moderate complexity, as here, you must implement `-makeWindowControllers` instead.

Replace the deleted `-windowNibName` method with this new `-makeWindowControllers` method:

```
- (void)makeWindowControllers {
    DiaryWindowController *controller =
        [[DiaryWindowController alloc] init];
    [self addWindowController:controller];
    [controller release];
}
```

You must also tell the compiler where to find the `DiaryWindowController` header file. Insert this line following `#import "DiaryDocument.h"`:

```
#import "DiaryWindowController.h"
```

The first statement of the `-makeWindowControllers` method differs from the first statement you wrote in `-makeWindowControllers` for the recipes document in Recipe 1. Here, after allocating memory for a `DiaryWindowController` object, you call its `-init` method, while in Recipe 1 you called the `RecipesWindowController` object's `-initWithWindowNibName:` method and supplied the name of the nib file.

The different approach you take here relieves the diary document of the need to know which nib file the diary window controller should load. The application specification indicates that the diary document, unlike the recipes document, displays its contents in a single type of window. All knowledge of how that window is set up should be encapsulated in the window controller itself, to reduce the interdependence between the diary document and the diary window controller. The document knows that it needs to create a diary window controller, but that is all it knows. The choice of nib files is an implementation detail that is best left to the window controller. You will write the necessary initialization code when you create the window controller in Step 3.

10. Close the `DiaryDocument` header and implementation file windows. You will now take a short detour to learn how to create a snapshot of your code from time to time in case you need to undo a mistake and return to its current state. In Step 3 you will return to the diary document and create its window controller.

Step 2: Save a Snapshot of the Project

You have started to develop the good habit of backing up your work at the end of every recipe, but you might want to keep track of your changes at intermediate stages too. In addition, it is often useful to compare your code now with your code as it existed an hour ago or a day ago, simply as a reminder of the changes you have made.

Xcode provides an easy way to do this, the *Snapshot* facility. At any time, no matter the state of your project, choose `File > Make Snapshot`. No dialog is presented, you don't have to name anything or figure out where to save anything, and you aren't interrupted with any confirmation dialog. It just happens, and it happens very quickly once Xcode has taken a few moments to set up the snapshot repository the first time you use it.

The snapshot repository is a disk image located in your Application Support folder at `~/Library/Application Support/Developer/Shared/SnapshotRepository.sparseimage`. Because it is a `sparseimage`, it automatically varies in size depending on its contents.

An Xcode snapshot is not intended to take the place of a source code management (SCM) facility. It is instead a means to keep track of the state of a project while you experiment with a development path that might not work out. In addition to allowing easy recovery, a snapshot gives you the ability to compare the code in the snapshot with your current experimental code or code in another snapshot, side by side. The point of it is to give you an easy way to examine recent changes to the project and to return to the point where you started the experiment, not to save your files against drive failure or accidental erasure. In effect, it is a project-wide undo and redo mechanism. Only your own work on your own computer is saved in a snapshot; it cannot be shared with other developers on a team, or even with another computer if you use two computers for development.

A snapshot saves the state of everything that you have saved in your project root directory. By default, the project root directory is the project directory. You can change it to a higher-level directory that contains related projects. You have no related projects at this time, so leave the project root directory at its default setting, the Vermont Recipes project folder. If you wanted to change it, you would open the information window for the project, select the General tab, and click the Configure Roots & SCM button.

1. In Xcode, make sure that everything in your project has been saved, and then choose File > Make Snapshot. If you didn't save everything, Xcode will ask you to do so. The only other feedback you see is that the File menu remains highlighted for a few moments while Xcode sets up the snapshot.
2. Choose File > Snapshots. The Vermont Recipes - Snapshots window opens. It contains a toolbar at the top with buttons to make another snapshot, to delete a snapshot, to restore the state of the project to the state saved in a snapshot, and to show or hide files. A single snapshot appears in what can become a long list of snapshots immediately below the toolbar. Xcode names the snapshot automatically based on what you were doing recently, and it adds a date stamp. In the bottom pane, you find fields to change the name of the snapshot and to enter comments to help you remember what this snapshot represents.
3. Change the name of this snapshot by selecting the text portion of the default name in the Name field, preceding the time stamp, and typing **Recipe 3 Step 1**.
4. In the Comments field, enter **After adding DiaryDocument but before adding DiaryWindowController** (Figure 3.2).



FIGURE 3.2 The Snapshots window with one snapshot.

5. Perform an experiment now to see how the Show Files feature works.
 - a. Select the DiaryDocument.h file, and then in an editing pane or window insert the line **REMOVE THIS LINE** below the `#import` directive and save the file.
 - b. Choose File > Snapshots.
 - c. In the Snapshots window, select the Recipe 3 Step 1 snapshot and click the Show Files button.

In the new pane that expands to the right, you see a list labeled Files Changed. It identifies DiaryDocument.h as the only file that was changed since you took the snapshot. If you had changed more files, all of them would be listed on the right. If you had taken more snapshots, all of them would be listed on the left, and you could select any two of them to compare their differences.

- d. Select DiaryDocument.h in the list of Files Changed.

A new pane appears below the Files Changed list showing the two snapshots of DiaryDocument.h side by side. The line you inserted is highlighted as a change (**Figure 3.3**). If you had made other changes, they too would be highlighted.

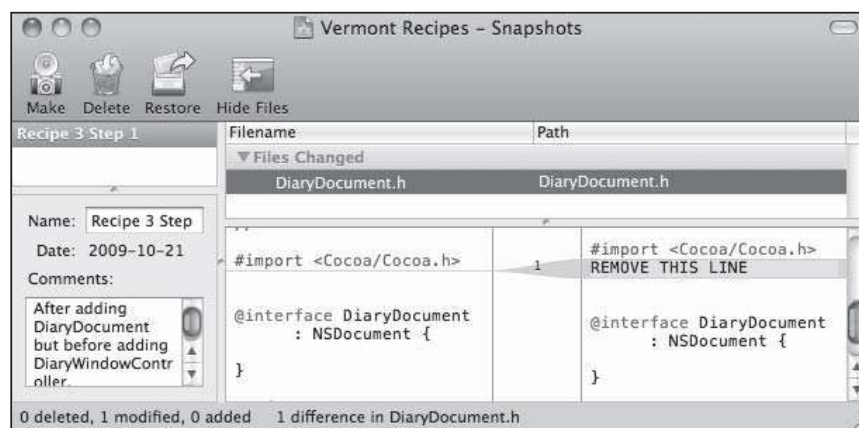


FIGURE 3.3
The Snapshots window expanded to show changes.

- e. In the list of snapshots on the left, select Recipe 3 Step 1 and click the Restore button. The comparison pane closes, the reference to DiaryDocument.h disappears from the list of files changed, and if you had the DiaryDocument.h window open, you saw that the line you added a moment ago disappeared from the file. The snapshot remains selected in the list, but a new snapshot has been added at the bottom of the list named Pre-Restore with a time stamp. You have no need to retain the Pre-Restore snapshot. Select it and click the Delete button, leaving only the Recipe 3 Step 1 snapshot.
- f. Close the Snapshots window and examine the DiaryDocument.h file's contents. The line you inserted, *REMOVE THIS LINE*, is now gone.

Because it is easy, you should get in the habit of taking a snapshot at the end of every step.

Step 3: Create the DiaryWindowController Class and Its Nib File in Interface Builder

You could create the new `DiaryWindowController` class by following the same steps you used in Step 1, but you should take this opportunity to learn how to create a new class in Interface Builder and merge it into the Xcode project. You can create the `DiaryWindowController`'s nib file at the same time. This demonstrates another dimension to the close integration of Xcode and Interface Builder.

In Step 7 of Recipe 1, you created the `RecipesWindowController` class to control the Vermont Recipes main window, and you made the window controller the File's Owner of the window's nib file. As a result, your `RecipesDocument` class no longer needed a nib file of its own. A document class should have nothing to do with the application's view other than to know how to communicate with its window controllers.

The same principles apply to `DiaryWindowController` and `DiaryDocument`.

1. Launch Interface Builder. If you installed it in the Dock, this requires nothing more than clicking its icon. Otherwise, find it in the Applications folder of your Developer folder and double-click its icon. The Choose a Template window opens. If you don't see the window, choose File > New.
2. In the Choose a Template window, select Cocoa in the left pane, and then select Window in the right pane. Click Choose, and an untitled nib file opens. You see the four windows you should by now expect: the main nib file window, an empty document window (this one does not contain the "Your document contents here" text field), and the Library and Inspector windows. The nib file is named Untitled, and it has not yet been saved.
3. Set up the File's Owner proxy. You know it should be a subclass of `NSWindowController`, just as the `RecipesWindowController` subclass you created in Recipe 1 owns the `RecipesWindow` nib file. Select the File's Owner proxy, and then choose the Object Identity inspector. You see that, by default, the File's Owner is `NSObject`, the root of almost all Cocoa classes. This is because you haven't yet created the `DiaryWindowController` class, and your new nib file doesn't yet know what kind of object it should be. Open the inspector's Class combo box, and you don't see anything called `DiaryWindowController` because it doesn't yet exist.

To create the `DiaryWindowController` class, turn to Interface Builder's Library window, the source of all new classes and objects in Interface Builder. Select its Classes tab, and then scroll down to `NSWindowController` and select it. In the pane at the bottom of the window, select the Lineage tab. You see the inheritance

hierarchy of `NSWindowController`, from its root class, `NSObject`, through the first subclass, `NSResponder`, to `NSWindowController` itself. You want to create one more level at the top of this hierarchy, `DiaryWindowController`, as a subclass of `NSWindowController`.

Click the Action menu at the bottom of the Library window, or Control-click (or right-click) `NSWindowController` in the list of existing classes. You see that the first menu item is New Subclass. Choose it.

4. In the New Subclass dialog, the “Add subclass of `NSWindowController` named” field contains a placeholder name, *MyWindowController*, and it is selected for editing. Type **DiaryWindowController** to replace it, select the “Generate source files” checkbox, and click OK.

A standard save file dialog opens, with the Save As name already set to `DiaryWindowController.m`. The Language pop-up menu is set to Objective-C, and the “Create '.h' file” checkbox is selected. Navigate to the Vermont Recipes project folder and click Save.

Yet another dialog appears, asking if you want to add the new files to the Vermont Recipes project. Before you examine this dialog, open the project window in the Finder. There you see that the new `DiaryWindowController` header and implementation files have been created and saved. However, they aren’t yet in the Xcode project.

5. Return to the Add Files to Project dialog that is still open in Interface Builder. Select the Vermont Recipes target and click Add. Out of the corner of your eye, you see the Classes list in the Library window scroll a little, and, looking over, you see that a new class, `DiaryWindowController`, has been added to the list and selected. In the Lineage pane at the bottom, you see that `NSWindowController` has a new subclass at the top of the hierarchy, `DiaryWindowController` (**Figure 3.4**).

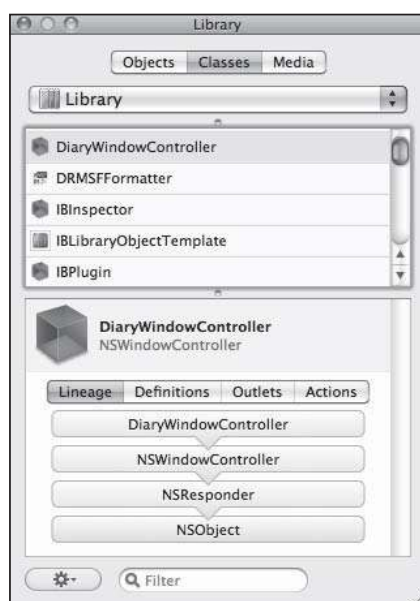


FIGURE 3.4
The Library window
after you have created
`DiaryWindowController`.

6. Now turn to the Xcode project window. There, probably near the bottom of the Vermont Recipes project group in the Groups & Files pane, you find `DiaryWindowController.h` and `DiaryWindowController.m`. Drag both of them up into the new Window Controllers subgroup of the Classes group that you created in Step 1.
7. Go back to the Object Identity inspector in Interface Builder, open the Class combo box, and scroll until you find `DiaryWindowController`. When you find it, select it, and your new `DiaryWindowController` object is now the File's Owner of the nib file.
8. For the `DiaryWindowController` class to be useful, you also have to make two connections. Select the File's Owner proxy and open the Diary Window Controller Connections inspector. To connect the `window` outlet that you find there, drag from the marker beside it (an empty circle) to the Window icon in the nib file's window. Then select the Window icon in the nib file's window, open the Window Connections inspector, and drag from the `delegate` outlet's marker to the File's Owner proxy.

The `window` and `delegate` outlets you just connected represent instance variables declared in `NSWindowController.h` and `NSWindow.h`, respectively. Later in this recipe, you will create some outlets of your own and connect them using the same technique.

9. The nib file is still untitled and unsaved, so save it now. Choose File > Save As, type **DiaryWindow** in the Save As field, and set the File Type to "Interface Builder Cocoa Document (XIB 3.x)." Read the "Nib File Formats" sidebar for information about various kinds of nib files. Remember that nib files should be saved in the `English.lproj` folder, or in another `lproj` folder if you're developing for another locale. All you have to do is navigate to the `English.lproj` subfolder of the Vermont Recipes project folder and click Save.

A now-familiar sheet appears, asking whether you would like to add the nib file to the Vermont Recipes project. Before dismissing it, look at the `English.lproj` subfolder of the project window in the Finder, and you see that the new `DiaryWindow` nib file has been created and saved.

10. Return to the sheet to add the nib file to the Xcode project. Select the Vermont Recipes target checkbox and click Add. The name of the nib file window changes to *DiaryWindow - English*.
11. Turn again to the Xcode project folder. The `DiaryWindow` nib file is now listed, probably near the bottom of the Vermont Recipes project group. Drag the nib file into the Resources group and drop it just under `RecipesWindow.xib`.

Nib File Formats

Interface Builder has saved its data in what are commonly called *nib* files since olden days. The format has changed a few times over the years, and it is important to choose carefully which format you use.

For Vermont Recipes, the choice is simple, because you are building it in Snow Leopard and it requires Leopard or newer to run. The *xib* format was introduced in Interface Builder 3.0 for use in Leopard or newer, and it is what you should use for development under these circumstances. Developers still refer to it as a *nib* file even though its file extension is *xib*. It was invented to provide closer integration with Xcode while developing a project. When you build the application, the *xib* files are compiled into *nib* files with the *nib* file extension. Unlike earlier versions, they cannot be read and edited by users who do not have access to your project files.

If you were editing in Leopard but building for Mac OS X 10.4 Tiger, you would use the 3.x *nib* format instead of the *xib* format. If you were editing and building in Tiger, you would use the 2.x *nib* format. See the *Interface Builder User Guide* for more information.

Unless you're writing plug-ins for Interface Builder, there isn't much more you need to know about *nib* file formats. When you localize your application, make the *xib* files available to your contractor in editable form, because they typically contain many strings intended for the user and therefore requiring translation.

12. Expand the Window Controllers subgroup of the Classes group, open the `DiaryWindowController` header and implementation files, and edit the identifying information at the top of each of them as you did in Step 4 of Recipe 1. You have to insert `// Vermont Recipes 2.0.0` below the name of the file. Interface Builder left this line blank because you created the file before you added it to the Vermont Recipes project.
13. You must now add a method implementation to initialize the window controller. At the end of Step 1, you overrode `NSDocument`'s `-makeWindowControllers` method in the `DiaryDocument` subclass. It called the `-init` method of the newly allocated diary window controller, instead of the `-initWithWindowNibName:` method you used in the recipes document in Recipe 1, in order to remove the decision about what *nib* file to load from the document and assign that task to the window controller. You must follow up now by providing a suitable implementation of `-init` in `DiaryWindowController`. If you don't, the application will call an inherited version of `-init`, possibly reaching all the way back to `NSObject`'s version of `-init`. The inherited `-init` method will know nothing about the diary window controller and its instance variables, and the diary window will not open because the window controller won't know how to find its *nib* file.

Implement the `-init` method in the diary window controller now, so that it can load the diary window nib file and open the diary window. First, do it in the simplest possible way in order to make it easy to understand:

```
- (id)init {
    self = [self initWithWindowNibName:@"DiaryWindow"];
    return self;
}
```

Initializing a newly allocated object in Cocoa is subject to a number of requirements. To understand them, it is important to read the “Allocating and Initializing Objects” section of Apple’s *The Objective-C 2.0 Programming Language*.

Initialization is primarily about setting the initial values of the object’s instance variables. If you provide no initial values, they are automatically set to `nil`, `NULL`, `0`, or `NO`.

An initialization method must call another initialization method inherited from the object’s superclass, directly or indirectly. You can count on the superclass’s initialization method to have set the superclass’s instance variables. Here, for example, the `-initWithWindowNibName:` method you call is inherited from the superclass, `NSWindowController`, and it is documented to turn on window cascading, set the `shouldCloseDocument` flag to `NO`, and set the autosave name to an empty string by calling `NSWindowController`’s designated initializer, `-initWithWindow:`. Only a class’s designated initializer is required to call the superclass’s designated initializer.

An initialization method must always return an object if initialization was successful or `nil` if it was not. The return type of the initialization method should be `id`. This allows you the greatest freedom to revise the application’s design later without having to rewrite the diary document class or other clients of the window controller. For example, you might someday want to create a more specialized kind of diary, and you might choose to do that by returning a different object of a different class in place of the object you just allocated.

An `-init` method should return `nil` if initialization was unsuccessful. The Cocoa frameworks rely on this convention, and you must support it. Here, if the call to the `-initWithWindowNibName:` method is unsuccessful, it returns `nil`, and you assign that result to `self`. Then, when your `-init` method returns `self`, it returns `nil`.

An initialization method should also always assign the newly initialized object to `self`. This is how Cocoa applications coordinate the interrelationships among the classes they use.

Another important requirement is that every class should have at least one designated initializer that is called, directly or indirectly, by all of its other initializers. For example, `NSWindowController`'s documented designated initializer is `-initWithWindow:`. Read the “Object Initialization and the Designated Initializer” sidebar for more information.

Object Initialization and the Designated Initializer

Initialization of Cocoa objects is governed by a well-defined convention that all Cocoa applications must follow. The convention is explained in some detail in the description of `NSObject`'s `-init` method in the *NSObject Class Reference*. `NSObject` is a fundamental class declared in the Foundation framework. Since most Cocoa classes inherit from `NSObject` (and the principal one that doesn't, `NSProxy`, nevertheless follows the `NSObject` protocol), reliance upon this convention is implicit throughout the Cocoa frameworks. If your application doesn't honor this convention, it probably won't work.

`NSObject`'s `-init` method does nothing except return an object. The method is available in every object that inherits from `NSObject`. Many such classes override the `-init` method and possibly provide one or more alternative initialization methods to do additional initialization. Because there can be many intermediate classes in the inheritance chain, a convention is needed to ensure that an appropriate initialization method of every object in the chain is called when a new object of any class is created. If initialization fails, a class's initialization method must release the object and return `nil` to signal failure.

To ensure that the initialization methods of classes intermediate between `NSObject` and the class are called, at least one of the initialization methods of the class must invoke an appropriate initialization method of its immediate superclass. This is known as the *designated initializer*, and every other initialization method in the class must ultimately call the class's designated initializer. If super's initializer returns a valid reference to the new object (as opposed to `nil`, indicating failure somewhere up the chain), then the new object knows that all classes above it in the chain have been successfully initialized, and it can go ahead with initialization of its own variables.

The designated initializer typically contains more parameters than any of the other initializers, allowing clients to pass as many unique values as possible to a new object. Other initializers are often provided in custom classes for special purposes, setting a variety of instance variables to default values and therefore requiring fewer parameters. Each of them must call the class's designated initializer, directly or indirectly, through a message to `self`, passing the default values and any parameter values in parameters of the designated initializer.

(continues on next page)

Object Initialization and the Designated_INITIALIZER *(continued)*

This guarantees that the variables of the class are initialized to appropriate values and that all intermediate classes higher in the hierarchy are initialized, and it avoids circular initialization references.

Typically, a class that inherits directly or indirectly from `NSObject` declares an `-init` method as well as alternative, more complicated initialization methods that take arguments to set the initial values of variables declared in the class. The initialization method that takes the most arguments—that is, the one capable of most completely setting up the object's initial state—is usually the designated initializer. Only the designated initializer actually sets any instance variable values; the others call the designated initializer to do this for them.

It is the developer's obligation to identify the designated initializer by a comment in the header file, to ensure that the developer of clients of the class knows which initialization method to call.

Now that you understand the basics, rewrite the `-init` method to use a more common technique:

```
- (id)init {
    if ((self = [self initWithWindowNibName:@"DiaryWindow"])) {
        // Insert any initialization code here.
    }
    return self;
}
```

This new version of `-init` achieves economy of code by relying on standard C coding techniques. Some developers regard this as tricky, or at least overly clever, but its use is sanctioned by a long history.

The portion of the first statement that lies inside the double parentheses is not unusual. It is an exact copy of the first statement in your initial, simple implementation of `-init`, assigning the result of `-initWithWindowNibName:` to `self`. It uses the assignment operator (`=`), not the equality operator (`==`). This is sometimes a source of confusion because the expression also operates as a Boolean test for the `if` statement.

The assignment statement is enclosed in parentheses, signifying that the result of the assignment is of interest. The result is either a pointer to the newly initialized object assigned to `self`, or it is `nil`. Either way, Objective-C can interpret the result as a Boolean value, YES or NO, respectively. To better appreciate what is happening, you could write it like this, but developers rarely do:

```
if ((self = [self initWithWindowNibName:@"DiaryWindow"]) != nil) {
```

or like this:

```
self = [self initWithWindowNibName:@"DiaryWindow"];  
if (self != nil) {
```

The point of testing for a `nil` result is to know whether it is safe to execute additional initialization code, which might rely on `self`'s having a value. You don't have any initialization code to insert at this point, so leave the `if` clause empty.

The outer parentheses represent a somewhat pedantic insistence on correct C syntax, which requires the test in every `if` statement to be placed within parentheses. I generally include both layers of parentheses as a reminder that this is an assignment operation combined with a Boolean test. Most developers use a single set of parentheses for simplicity, and this works unless you set the compiler's *pedantic* flag.

14. Close the `DiaryWindowController` header and implementation file windows for the time being. You will add more methods to it later.
15. Make a new snapshot using the techniques you learned in Step 2. If you didn't already save all of the files you created in this step, Xcode will ask you to save them now. Then rename the snapshot **Recipe 3 Step 3** and enter the comment **After adding `DiaryWindowController` and `DiaryWindow.xib`**. I won't remind you to make a snapshot again, so remember to do it at the end of every step.

Step 4: Add Scrolling Text Views to the Diary Window

In the detailed specification of the Chef's Diary at the beginning of this recipe, you provided for a window with two scrolling text views embedded in the panes of a split view with a movable divider. You created the window controller and the nib file using Interface Builder in Step 3. In this step, you continue working in Interface Builder to add the split view and the scrolling text views.

1. Open the new `DiaryWindow` nib file in Interface Builder.
2. Select the Window icon in the main nib file window, and then examine the Window Attributes inspector. There is no need to give the window a name, because you will provide a name in code. To avoid confusion, it might therefore be best to remove the default name `Window` in the Window Attributes inspector. The other settings are already appropriate for a general-purpose window, providing, for example, close and minimize buttons in the title bar and a resize control at the bottom right corner. However, deselect the Shows Toolbar Button

checkbox, since the diary window does not have a toolbar. Cocoa is smart enough not to show the toolbar button in a window that doesn't have a toolbar, but it is again a good idea to avoid confusion.

3. Select the Window Size inspector. You can set the default size of a new window here, as well as its starting position on the screen. For now, don't worry about getting it exactly right. It's too early to know how much real estate the window will need because you haven't yet added any user interface elements.
4. Turn to the Window Identity inspector. You see that the class of the window is `NSWindow`. This is appropriate. It is rarely necessary to subclass `NSWindow`, because it declares many delegate methods that you can use to control its behavior.

There is one interesting field at the bottom of the Window Identity inspector, labeled Notes. Enter some text here that will help anybody looking at this file later (including you) to understand what it is. Enter **Free-form RTF text for the Chef's Diary**. Leave the Show With Selection checkbox deselected, so that you won't have to look at this note in a help tag every time you select the window in Interface Builder while you're working on it.

5. In the Library window, select the Objects tab, and then navigate to Library > Cocoa > Views & Cells > Layout Views. Drag the Horizontal Split View into the document window, and then position and resize it to fill the window except for some space across the bottom. Don't be concerned yet about how much space is needed at the bottom for the controls. Just allow a little more room than really seems necessary. You'll add the controls and adjust the size and position of everything in Recipe 4.
6. In the Split View Attributes inspector, leave the Style pop-up menu set to "Pane splitter." You want the divider in this window to be visible even when the user collapses the split view to a single pane by dragging the divider all the way to the top or bottom, and the standard splitter with a dimple is therefore appropriate.

In Snow Leopard, Apple's *Human Interface Guidelines* suggest that the thin divider is preferred. However, when a pane can be collapsed all the way so that the divider is hidden, you have to supply a button to expand it again. In Vermont Recipes, you will follow the example of Apple's Mail application, where the "Pane splitter" divider is used. The divider is visible at all times, so you don't have to add a button.

7. Next, fill the top and bottom panes of the split view with scrolling text views. There are two ways to do this, but one is decidedly more powerful and at the same time simpler than the other.

The hard way is to do it piecemeal. This also creates a less powerful end result, so you might not want to follow along with this. If you do follow along, create a snapshot now so that you can restore the current state of the project to get rid

of this false start. Go back to the Layout Views section of the Library window and drag a Scroll View into the top pane of the window's split view. Do the same with the bottom pane. Each scroll view comes with an embedded custom view. Select each custom view in turn and use the View Identity inspector to change its class from `NSView` to `NSTextView`. Look at the Text View Attributes inspector, and you see that it doesn't provide many settings. If you continue with this setup, you will have to do almost everything in code to configure the RTF text editing features of the Chef's Diary. If you followed along, open the Snapshots window and restore the project to the snapshot you made a moment ago. If you didn't take a snapshot, manually delete the scroll view and its embedded `NSTextView` from each pane of the split view. Now you're ready to start over.

This time, do it the easy but powerful way. Use the Library window to navigate to Library > Cocoa > Views & Cells > Inputs & Values. Scroll down until you find the Text View. Drag one text view into the top pane and another into the bottom pane of the split view.

These text views may be a little too large if your diary document window is too small. If so, you should make the window larger or drag the bottom edge of the split view up or down to make the panes larger. Hold down the Command key while you drag the bottom edge of the window to ensure that embedded views resize to match.

Once you can see each text view in its entirety, drag it into the top-left corner of its pane and drag its resize handle to its pane's bottom-right corner to fill the pane.

8. In each scroll view in turn, select the Scroll View Attributes inspector and change one setting. The scroll views will hold free-form text, so it is appropriate to leave the Show Horizontal Scroller checkbox deselected and the Show Vertical Scroller checkbox selected, as you find them. You will learn later that the text views come with built-in behaviors that make a horizontal scroll bar inappropriate. However, the window will appear less busy if the vertical scroll bar is hidden when not needed, so select the Automatically Hide Scrollers checkbox in the Scroll View for both panes of the Horizontal Split View. When you do this, the placeholders for the vertical scroll bars in the document window disappear (**Figure 3.5**).

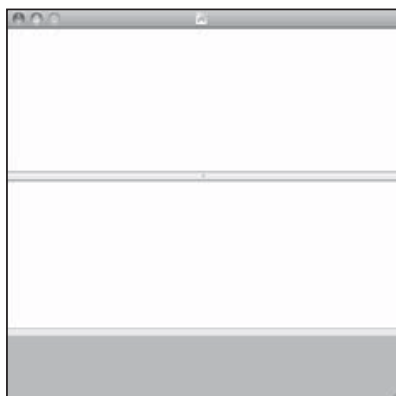


FIGURE 3.5 The diary window with scrolling text views in a split view.

9. Select the text view that is embedded in the scroll view in the top pane of the split view. To do this, you have to click twice. If you start with the window or the split view selected, the first click in the top pane selects the embedded scroll view. You have to click a second time, near the top of the pane, to select the embedded text view. You can tell which is selected by looking at the current title of the Inspector. The easiest way to select the text view is to Shift-Control-click and choose it from the contextual menu.

With the embedded text view selected, open the Text View Attributes inspector (**Figure 3.6**). The settings include just about everything you could ask for in a text editor, and all of these features come for free with no coding required on your part. It even includes the same ruler you see in Apple's TextEdit application. One feature that is particularly astonishing is the Rich Text checkbox. Simply by your leaving it selected, the Chef's Diary gains all the formatting capabilities of RTF. The settings as you see them reflect a good balance between power and ease of use, so leave them as they are.

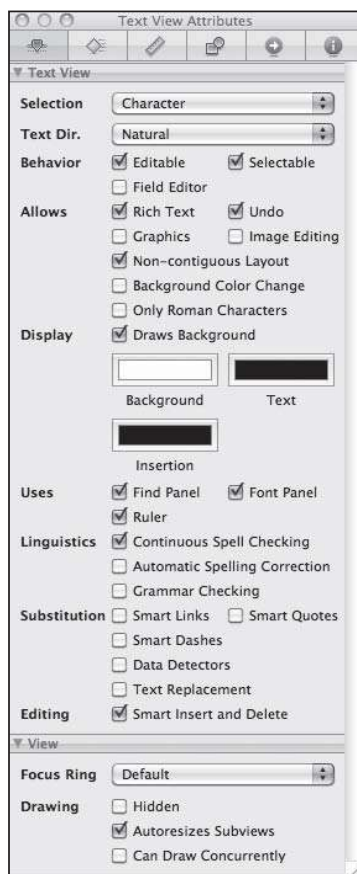


FIGURE 3.6 The Text View Attributes inspector.

10. Now that the split view and its embedded subviews are in place, it is time to set up their autosizing behavior.
 - a. Start with the innermost embedded subviews, the text views in the top and bottom panes. Select one of them and go to the Text View Size inspector. You find all of the springs and struts enabled, meaning that the size of the

text view is flexible horizontally and vertically, and all four of its edges are locked to the edges of the enclosing scroll view. Both text views are perfect.

- b. Now turn to the containing scroll views, which are embedded in the top and bottom panes of the split view. Select one of them, and you immediately see that the autosizing settings are wrong. The scroll view should always fill the upper or lower pane of the split view as the user resizes the window or moves the divider up or down to resize the panes relative to one another. Enable all of the springs and struts in both scroll views.
- c. Finally, turn to the outermost split view, which is embedded in the window. The Split View Size inspector shows that its springs and struts are also wrong. Again, enable all of them.
- d. Test the autosizing behavior of the new views. Choose File > Simulate Interface. Resize the document window continuously, and you see that the split view resizes continuously to match, always filling the window except for the space at the bottom where you will soon add several controls.

The space at the bottom of the window remains of constant height, instead of resizing proportionately as did the left pane of the split view in Recipe 2. This is because the space at the bottom of the window is not part of the split view. In fact, the two panes of the split view do resize proportionally, as you expect and desire in this context. Move the divider up and down, all the way to the bottom and top of the split view. It remains in place as you resize the window. Quit the Cocoa Simulator.

11. A split text view should normally show only one pane, leaving it to the user to drag the divider upward or downward to see the other pane only when desired. In the document window in Interface Builder, Command-drag the divider to the bottom of the split view.

Step 5: Create the VRDocument-Controller Class and a New Menu Item

You haven't yet arranged for storage of the document's contents. Before you can attend to that, however, there is one other important feature missing. You have provided no way to create a new Diary Document or to open its window.

You know from Recipe 1 that the Cocoa document-based application template automatically provides a mechanism to create new documents of the application's primary document type. In Vermont Recipes, the primary document type is the

recipes database, controlled by `RecipesDocument`, `RecipesWindowController`, and the `RecipesWindow` nib file, all of which were provided to you by the template under the name *MyDocument*. The application's File menu came with a standard New menu item in the `MainMenu` nib file provided by the template, and you saw in Recipe 1 that you could use it to open new recipes document windows at will. The first new recipes document is automatically opened when you launch the application.

Before devising a way to create a new Chef's Diary document and open its window, explore how the application was able to create a new recipes document. Open the `MainMenu` nib file and double-click the `MainMenu` icon. Interface Builder opens a window showing a mockup of the Vermont Recipes menu bar. Click the File menu in the mockup to open it, and then click its New menu item to select it. Open the Menu Item Connections inspector, and there you see, in the Sent Actions section, that a `newDocument:` action message is associated with the First Responder proxy. To learn what it does, search for the `-newDocument:` method in the Xcode Documentation window.

You find the entry for the `-newDocument:` method in the *NSDocumentController Class Reference*. Evidently, Cocoa treats an instance of `NSDocumentController` as the first responder for the New menu item when the user opens the File menu. You will learn more later about the Cocoa *responder chain* that lies behind the nib file's First Responder proxy. For now, just note that `NSDocumentController` does not inherit from Cocoa's `NSResponder` class. Interface Builder is nevertheless able to connect the New menu item and a number of others to action methods in `NSDocumentController` because, according to Apple's *Document-Based Applications Overview*, `NSDocumentController` is "hard-wired to respond appropriately."

You learn in the `-newDocument:` method's description that it opens an untitled document whose type is the first document type specified by the `CFBundleDocumentTypes` key in the application's `Info.plist` file. You also learn that the subclass of `NSDocument` that you associate with the first document type in `Info.plist` is the subclass that the `-newDocument:` method opens. Recall from Step 9 of Recipe 1 that you set the Core Data XML document type as the first document type in the `Vermont_Recipes-Info.plist` file and that you specified `RecipesDocument` as the subclass of `NSDocument` that is associated with the XML document type.

Thus, as the documentation tells you, the Vermont Recipes application's New menu item tells the `-newDocument:` method to call the `-[NSDocumentController openUntitledDocumentAndDisplay:error:]` method. Look up that method in the documentation, and you learn that it calls `-[NSDocumentController makeUntitledDocumentOfType:error:]` with the first parameter set to the document type associated with the first document. That is `@public.xml` in Vermont Recipes. The `-openUntitledDocumentAndDisplay:error:` method then, according to the documentation, goes on to call other Cocoa methods that add the appropriate window controller to the list of active window controllers and open the window.

That is all well and good for an application that recognizes only a single type of document and implements only one subclass of `NSDocument`. Cocoa's built-in `NSDocumentController` class handles such an application properly without subclassing, by means of the convention it implements whereby the first document type listed in the `Info.plist` file governs. For this reason, the Cocoa documentation tells you that you normally don't have to subclass `NSDocumentController`. What the documentation doesn't tell you is that, in some cases, you do have to subclass `NSDocumentController`. Vermont Recipes is one of those cases.

If you look further, you find that Apple's *Document-Based Applications Overview* explains how to open an auxiliary document in a more complex application that supports multiple types of documents, as does Vermont Recipes. Turn to the "Creating Multiple-Document-Type Applications" section of the document. There you learn that you must subclass `NSDocumentController` in this situation, and several other requirements are laid out for you. Review them now, because they provide a roadmap for what you have to do. Bear in mind that you wouldn't have to do any of this if Vermont Recipes implemented only a single kind of document, whether it were the recipe document alone or the diary document alone, because `-[NSDocumentController newDocument:]` would then work correctly out of the box.

You have already implemented some of the requirements outlined in the *Document-Based Applications Overview*. You have created new subclasses of `NSDocument` and `NSWindowController` as well as a new nib file for the Chef's Diary window. You have also already made the `DiaryWindowController` object the File's Owner of the nib file, connected its `window` outlet to the nib file's document window, and connected the window's `delegate` outlet to the File's Owner proxy.

Now you must implement the remaining requirements. Start by subclassing `NSDocumentController`. In it, write a new action method to fill the same role for a Chef's Diary document that is played by `-newDocument:` for the primary recipes document type. Finally, provide a suitable user interface element, such as a menu item, and connect the new action to it so that the user can send the action. Take care of all of these requirements in this step. In the next step, you will add the new document type to the `Info.plist` file and associate it with the new `DiaryDocument` subclass.

1. Return to Apple's *NSDocumentController Class Reference* and find the discussion of the class method `+sharedDocumentController`. There you learn that calling this method instantiates and returns a new instance of the application's shared document controller, but only if one does not already exist. In other words, if you call the method a second time, it returns a reference to the first shared document controller and does not create a second. You can take advantage of this in your applications by calling the method from any object where you need access to the application's documents. You don't have to save a reference to the shared document controller, because a class method is global; you can call it anywhere

simply by targeting the class itself as the receiver of the method. The documentation goes so far as to caution against saving a reference to the shared document controller, admonishing you to always use the `+sharedDocumentController` method instead.

Although the Cocoa frameworks include other examples of shared singleton classes, `NSDocumentController` is unusual in that every document-based application instantiates it automatically when you launch the application. Because this happens so early in the life of the application, you must take special care to get your licks in first if you want to subclass it.

In the “Creating a Subclass of `NSDocumentController`” section of the *Document-Based Applications Overview*, Apple offers two ways to ensure that the `+sharedDocumentController` method always returns your subclass of `NSDocumentController`. One is to instantiate it in the application’s main nib file, which in Vermont Recipes and most other applications is the `MainMenu` nib file. The application always loads the main nib file early in the launch procedure, before it instantiates its own shared document controller. The other is to instantiate it by implementing `NSApplication`’s `-applicationWillFinishLaunching:` delegate method and coding it to create an instance of your document controller subclass. The application never creates its own shared document controller before it calls this delegate method. Either way, when your code calls `+sharedDocumentController` thereafter, it will always return the instantiated subclass instead of creating a new `NSDocumentController` object. The fact that the type of the subclass differs from the type of the superclass, `NSDocumentController`, is not a problem because the return type of `+sharedDocumentController` is `id`, precisely to support subclassing.

It is time to create your subclass of `NSDocumentController`, which you name *VRDocumentController*. Create the `VRDocumentController` class header and implementation files, and instantiate the new document controller in the `Main Menu` nib file.

- a. Use Xcode to create the files. By now, you know the drill. Choose `File > New File`. In the `New File` window, select `Cocoa Class` in the left pane and “Objective-C class” in the upper-right pane. In the lower-right pane, use the “Subclass of” pop-up menu to choose `NSObject`. The menu does not include `NSDocumentController` as one of its choices because subclassing `NSDocumentController` is relatively rare. You will change the superclass momentarily in the header file. Click `Next`.
- b. In the next window, enter **`VRDocumentController`** in the `File Name` field to create `VRDocumentController.m`, and leave the “Also create ‘`VRDocumentController.h`’” checkbox selected. Set the `Location` to the Vermont Recipes project folder; set the project to add it to Vermont Recipes; select the Vermont Recipes target checkbox; and click `Finish`. Drag the new

files to an appropriate place in the Classes group in the Groups & Files pane. I generally place important files that stand apart conceptually from any subgroup at the top of the Classes group.

- c. Open both VRDocumentController files and change the identifying information at the top following the model of Step 4 of Recipe 1.
 - d. In the `@interface` directive near the top of the VRDocumentController.h header file, change the superclass of VRDocumentController from NSObject to **NSDocumentController**.
 - e. Save both files.
2. Instantiate the singleton VRDocumentController object in the MainMenu nib file. First, open MainMenu.xib in Interface Builder. Then, in the Library window, select the Classes tab and scroll down to the bottom. There, along with your RecipesDocument and RecipesWindowController classes, you see the new VRDocumentController class. Drag it into the MainMenu.xib window. Now, when a user launches the Vermont Recipes application, it will load the MainMenu nib file and the instantiated VRDocumentController subclass. From now on, the `+sharedDocumentController` class method will return the subclass (**Figure 3.7**).



FIGURE 3.7 The MainMenu.xib file with a Document Controller object added.

3. Go back to the VRDocumentController code files and add a custom action method to open a new Chef's Diary document. Name it `-newDiaryDocument:`. You implement it using logic similar to that described in the documentation for the built-in `-newDocument:` method.
 - a. In the VRDocumentController.h header file, enter this declaration of the method:


```
- (IBAction)newDiaryDocument:(id)sender;
```
 - b. In the VRDocumentController.m implementation file, enter this implementation of the method:


```
- (IBAction)newDiaryDocument:(id)sender {
    DiaryDocument *diary =
        [self makeUntitledDocumentOfType:
         @"com.quecheesoftware.vermontrecipes.diary"
         error:NULL];
```

```

        if (diary != nil) {
            [self addDocument:diary];
            [diary makeWindowControllers];
            [diary showWindows];
        }
    }
}

```

The first statement calls the `-makeUntitledDocumentOfType:error:` method that `VRDocumentController` inherits from its superclass, `NSDocumentController`. You pass the string object `@com.quecheesoftware.vermontrecipes.diary` in the first parameter. This is the same string you will enter in Step 6 as the value of the `LSItemContentTypes` key for the diary document in the `CFBundleDocumentTypes` entry in the `Info.plist` file. The `-makeUntitledDocumentOfType:error:` method uses this value to look up the associated value of the `NSDocumentClass` key in the `Info.plist` file. The value is `DiaryDocument`. The method then instantiates and initializes a Chef's Diary document for you. The method returns a reference to the document, which you assign to the `diary` local variable. Pass `NULL` in the second parameter for now; you'll learn about the `error` parameter that exists in many Cocoa methods later.

If the method successfully creates a new diary document and the `diary` local variable therefore is not `nil`, the second statement tells the document controller, `self`, to add the new document to the list of documents that it maintains in order to fulfill its many responsibilities as the application's global document controller.

The third statement tells the new document to instantiate and initialize a new window controller for the document. You implemented an override of `NSDocument`'s `-makeWindowControllers` method in Step 1 because the document requires a subclass of `NSWindowController` and the simple `-windowNibName` method of `NSDocument` is inadequate in this context.

The fourth statement opens the Chef's Diary window, bringing it to the front and making it the application's key window—that is, the window that is ready to receive input from the keyboard.

- c. If you're on your toes, you realize that you specified `DiaryDocument*` as the type of the `diary` local variable in the `-newDiaryDocument:` method's implementation, but `VRDocumentController` doesn't know anything about the `DiaryDocument` class. Add this preprocessor directive immediately following `#import "VRDocumentController.h"` in the `VRDocumentController.m` implementation file:

```
#import "DiaryDocument.h"
```

- d. I recommend that you make one additional change. The first statement in your new `-newDiaryDocument:` method passes the string object `@com.quecheesoftware.vermontrecipes.diary` to the `-makeUntitledDocumentOfType:error:` method. You might need to use the same string in other parts of the application, but it will be very easy to introduce a typographical error. If you do, the compiler won't notice it, so you will be faced with a difficult debugging task down the road. One way to solve this problem is to use the standard C `#define` preprocessor directive to define it at the top of the file. Then you only have to type it once.

Near the top of the `VRDocumentController.m` implementation file, below the two `#import` directives, add this:

```
#define DIARY_DOCUMENT_IDENTIFIER
    @"com.quecheesoftware.vermontrecipes.diary"
```

Then revise the first statement in the `-newDiaryDocument:` method to use the definition, like this (Figure 3.8, on the next page):

```
DiaryDocument *diary =
    [self makeUntitledDocumentOfType:DIARY_DOCUMENT_IDENTIFIER
    error:NULL];
```

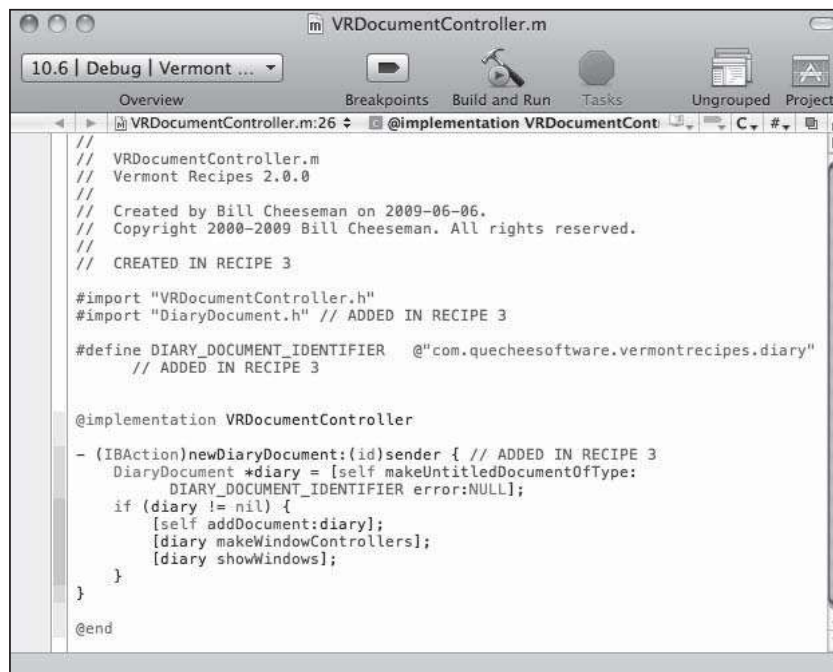


FIGURE 3.8
The `VRDocument.m` implementation file with a new method.

4. Finally, you're ready to create a user interface element that sends your new action method to the document controller to create a Chef's Diary document and open its window. A commonly used UI element to perform this action is a second New menu item in the File menu, named so as to distinguish it from the existing New menu item. Xcode itself follows this pattern, with New Project and New File items at the top of the File Menu. Do it now to get a foretaste of what

you can do in Interface Builder with the application's menu bar. You will do more with it in Recipe 5.

- a. Open the MainMenu nib file, and then double-click the MainMenu icon to open a mockup of the menu bar.
- b. Click the mockup's File menu to open it.
- c. Double-click the New menu item to select its title for editing, and change it to **New Recipes File**. Leave its sent `newDocument:` action connected to the First Responder proxy in the Menu Item Connections inspector.
- d. In the Objects pane of the Library window, choose Library > Cocoa > Application > Menus. Drag a Menu Item object into the MainMenu window and drop it immediately below the New Recipes File menu item in the open File menu. Double-click it and change its title to **New Chef's Diary**.
- e. Control-drag from the New Chef's Diary menu item to the First Responder proxy in the MainMenu nib file's window. In the Received Actions HUD, choose the `newDiaryDocument:` action.

You could just as well have connected the action directly to the Document Controller icon, since the action method is implemented there. Using the First Responder instead gives you greater freedom to revise the application's architecture later—for example, by deleting the `VRDocumentController` subclass from the nib file and instantiating it in another fashion.

- f. Save the nib file.
5. Build and run the application now to test what you've created up to this point. Click the Build and Run icon in the Xcode project window's toolbar. Click Save All if you're told there are unsaved changes, and wait a few moments.

Open the File menu, and you see the two New menu items, New Recipes File and New Chef's Diary. You have to stop testing now, because you haven't yet set up the Info.plist file to recognize the new `DiaryDocument` class. You'll edit the Info.plist file next, and then do some full-scale testing.

Step 6: Add the Diary Document to the Info.plist File

You set up the application's Info.plist file in Step 9 of Recipe 1, including entries for the recipes document's primary document type. Now you must add entries to the Info.plist file for the diary document. In addition to document type information similar to what you provided in Recipe 1, here you create a unique Uniform Type Identifier for the diary

document that tells the Finder and other applications that Vermont Recipes owns documents of this type. As a result, double-clicking a saved diary document's icon on the desktop will launch Vermont Recipes and open the document in its diary window.

1. Start by adding information about the `DiaryDocument` class to the `CFBundleDocumentTypes` item that you created in the `Info.plist` file in Recipe 1.
 - a. Open the `Vermont_Recipes-Info.plist` file in Xcode and, using the contextual menu in its property list editing window, choose `Show Raw Keys/Values`.
 - b. Expand the `CFBundleDocumentTypes` item and select the second and last item in the array, Item 1. Open the contextual menu on it and choose `Add Row`, or click the `Add (+)` button to its right. A new row appears with the key `"Item 2 ()"`. Expand it, and you see that it comes with two dictionary elements having the keys `CFBundleTypeName` and `LSHandlerRank`. The empty parentheses in the key are explained by the presence of the `CFBundleTypeName` key without a corresponding value. You need to change these two entries and add several additional keys.
 - c. With Item 2 selected and expanded, you see a button with an outline icon, extending into the margin to the right of Item 2. Click the button to add a new subentry at the top of the list of subentries. Choose `CFBundleTypeRole` as its key from the combo box that opens in the `Key` column, and then choose `Editor` in the `Value` column from its pop-up menu. Choosing `Editor` as the application's role means that Vermont Recipes is able to create applications of this type as well as read them.
 - d. Select the `CFBundleTypeName` entry and enter **Vermont Recipes Diary** as its value.
 - e. Select the `LSHandlerRank` entry and change its value from `Default` to `Owner`. This signifies that Vermont Recipes is the application that should open when the user double-clicks a diary file that was created by Vermont Recipes.
 - f. Click the `Add (+)` button to the right of the `LSHandlerRank` entry, and then choose `LSItemContentTypes` as the new subelement's key. Expand it and enter the value **com.quecheesoftware.vermontrecipes.diary** for Item 0. You will shortly define this content type as conforming to `public.rtf`, and you will export it so that the Finder and other applications recognize it.

This is the same UTI you specified in Step 5 as the document type that the `-newDiaryDocument` action method should open. That action method tells Cocoa to look up the UTI under the `LSItemContentTypes` key of the `Info.plist` file, in order to find what document subclass is associated with it. You will specify that association in a moment.

You don't have to use the application's `CFBundleIdentifier` as the base for the content type UTI. Here, the bundle identifier is `com.quecheesoftware.Vermont-Recipes` because of the way you specified it in Step 9 of Recipe 1, but you have specified the `LSItemContentTypes` entry by appending *diary* to `com.quecheesoftware.vermontrecipes.` with different capitalization and no hyphen. Later, you will append *recipes* to this value to form a UTI for the recipes document.

It is not necessary to use lowercase letters exclusively, as you have done here, and many applications do not. The legal characters in a UTI are the uppercase and lowercase letters of the Roman alphabet, the numerical digits 0 through 9, the dot (`.`), and the hyphen (`-`). UTIs are case-insensitive. In fact, if you specify a UTI with uppercase letters, the system will sometimes send it back to you with lowercase letters. I prefer to use lowercase letters from the outset because I find it less confusing.

- g.** Collapse the new `LSItemContentTypes` entry, click the Add (+) button to its right, and choose the `NSDocumentClass` key. Enter **DiaryDocument** as its value. You anticipated the need to do this when you created a subclass of `NSDocumentController` in Step 5 and added a custom action method to it. This entry associates the `DiaryDocument` class with the `public.rtf` content type so that the application will open a document using the correct Vermont Recipes document subclass.
- h.** Click the Add (+) button to its right and enter **NSExportableTypes**. Set its type to Array and expand it. Enter **com.quecheesoftware.vermontrecipes.diary** as the value of Item 0. This is the same value that you used in the `LSItemContentTypes` entry.

You don't have to add an `NSPersistentStoreTypeKey` entry, as you did in Recipe 1, because the diary document is not a Core Data file.

- 2.** Go through the same process to create a fourth entry in the `CFBundleDocumentTypes` array. This entry will tell the Finder and other applications that Vermont Recipes can open and edit any RTF file, even if the file was created in some other application and has nothing to do with cooking. This is a convenience that allows users to create a document in any RTF-capable application, such as TextEdit, and open it in Vermont Recipes if they think it suitable.

 - a.** With Item 2 of the `CFBundleDocumentTypes` array selected, add a new type, Item 3. If Item 2 is collapsed, do this by clicking the Add (+) button to its right. If Item 2 is expanded, hold down the Option key to change the outline button to an Add (+) button.
 - b.** Choose the `CFBundleTypeRole` entry and choose Viewer as its value. This signifies that Vermont Recipes can open any RTF file. It can even edit any

RTF file, but it can only save the edited file as a `com.quecheesoftware.vermontrecipes.diary` file with the *vrdiary* file extension. The Save menu item is disabled when a generic RTF file is active in the diary window.

- c. Choose the `CFBundleTypeName` entry and enter **Vermont Recipes Diary**.
- d. Create a new entry and choose `LSHandlerRank` as its key. Change its value from Default to Alternate. This signifies that Vermont Recipes is not the application that should open when the user double-clicks a generic RTF file that was not created by Vermont Recipes.
- e. Create a new entry and choose `LSItemContentTypes` as its key. Expand it and enter the value **public.rtf** for Item 0. As a result, the Finder and other applications will know that Vermont Recipes can open any RTF document. When the user drags a generic RTF file over the Vermont Recipes application icon, the icon will highlight to indicate that it can handle the file.
- f. Create a new entry and choose `NSDocumentClass` as its key. Enter **DiaryDocument** as its value. This associates the `DiaryDocument` class with the `public.rtf` content type in order to open a generic RTF document using the Vermont Recipes `DiaryDocument` class.

You don't need an `NSExportableTypes` key for the generic RTF type, because `public.rtf` is created and recognized by the system. Only Apple can create keys in the *public* domain (**Figure 3.9**).

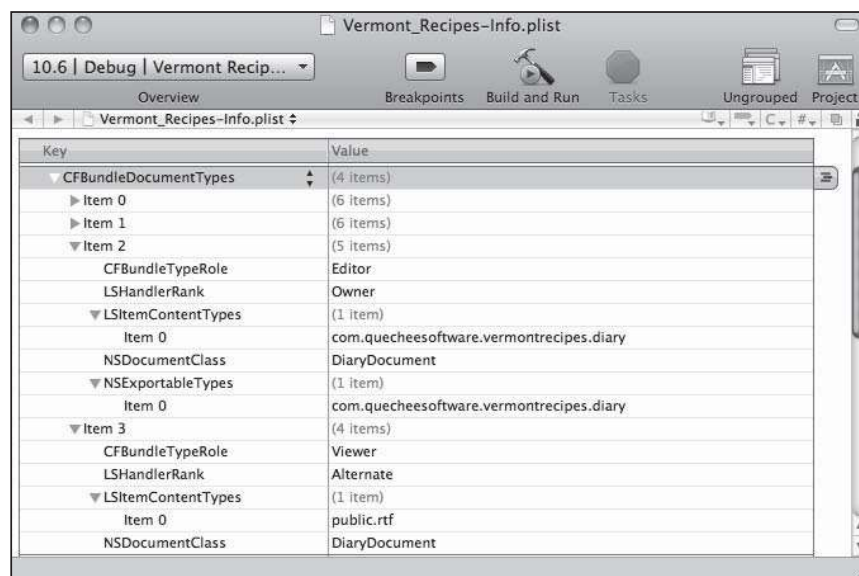


FIGURE 3.9
The Info.plist file showing the new document type.

3. One thing remains to complete the Info.plist document type infrastructure for the application. You marked the `com.quecheesoftware.vermontrecipes.diary` document type as exportable to the Finder and other applications, but you must declare the UTI to make it usable. The Finder and other applications must know several things about a unique document type owned by an application.

You declare an exportable UTI in a `UTExportedTypeDeclarations` entry in the `Info.plist` file. The entry is a dictionary with several keys.

- a. Start by selecting the last entry of the `Info.plist` file, `NSPrincipalClass`, and clicking the Add (+) button to create a new entry below it. Choose `UTExportedTypeDeclarations` as its key, and expand it to begin adding subentries.
- b. Create the first subentry and choose `UTTypeConformsTo` as its key. Expand it and enter **public.rtf** as the value of Item 0.
- c. Create a second subentry in Item 0 of `UTExportedTypeDeclarations` and choose `UTTypeDescription` as its key. Enter **Vermont Recipes Diary** as its value. The Finder displays this value in the Kind field of its Get Info window for any document of this type.
- d. Create a third subentry and choose `UTTypeIdentifier` as its key. Enter **com.quecheesoftware.vermontrecipes.diary** as its value. The Finder and other applications will recognize this as the exported type declaration for every Vermont Recipes diary document, and they will be able to associate the other values with every such document. For example, they will know that a Vermont Recipes diary document conforms to the `public.rtf` type.
- e. Finally, create a fourth subentry and choose `UTTypeTagSpecification` as its key. Expand it and create a subentry. You have to enter the key for this manually. Enter **public.filename-extension** as its key and enter **vrdiary** as its value. This tells Vermont Recipes to add *vrdiary* as the file extension of every diary document file it creates using the Save As menu item. You can add other tags, such as MIME types, if you wish (**Figure 3.10**).

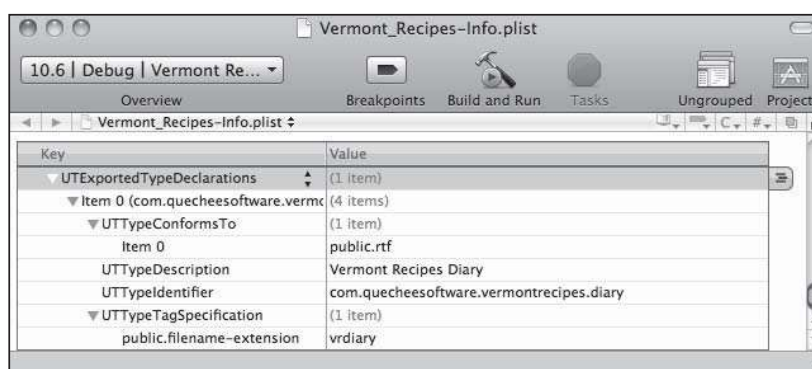


FIGURE 3.10
The `Info.plist` file showing the new exported document type.

4. One of the keys you added to the `Info.plist` file is displayed to users, so it must be added to the `InfoPlist.strings` file to permit localization. The `UTTypeDescription` field holds the value **Vermont Recipes Diary**, which the Finder shows in the Kind field of the Get Info window.

In InfoPlist.strings, add this entry at the bottom:

```
"Vermont Recipes Diary" = "Vermont Recipes Diary";
```

Do the same for the Vermont Recipes Database UTTypeDescription. Just above the Vermont Recipes Diary entry you just added in InfoPlist.strings, add this entry:

```
"Vermont Recipes Database" = "Vermont Recipes Database";
```

It is common for strings files to use the string itself as the key, as you've done here, if you are developing in the English locale. It is turned into a hash table in the finished application, so speed is not an issue. Even in the English locale you can see it at work by providing a different value for the key. Try entering the value of the Vermont Recipes Diary key as **New Hampshire Recipes Diary**. When you build and run the application and save a diary document, the Finder's Get Info window reports its kind as New Hampshire Recipes Diary.

5. Now you can do some serious testing. Build and run the application. Open the File menu and choose the New Chef's Diary menu item. A new window opens containing an empty RTF text view.

Hold on to your hat, because a big surprise is coming. Click in the window's text view and start typing. It works! Believe it or not, you have created a very powerful text editor while writing hardly any code. Explore what you can do with it.

First, type some text, and then press the Return key and type additional text to create as many lines of text as it takes to fill the text view. As soon as the insertion point reaches the bottom of the pane, a vertical scroll bar appears, and you can use it to scroll up and down. If you delete some lines or drag the window's resize control to make the window taller, the scroll bar disappears again as soon as the last line of text comes into view.

Select one of the lines of text, and then choose Format > Font > Bold. The Format menu works, and the selected line appears in boldface. Select a word or phrase in another line and choose Format > Font > Underline. It works, too. Choose Format > Fonts, and in the system Fonts panel select an interesting font, such as the new Chalkduster font added to Snow Leopard. Select another line and choose Format > Text > Align right. Choose Format > Text > Show Ruler. The standard system ruler appears, and its controls work.

Do some typing, and then choose Edit > Undo Typing. You see that Undo and redo work as expected in a finished text editing application. Select a line, choose Edit > Cut, and then move the insertion point and choose Edit > Paste. You see that cut and paste work as expected. Double-click a line and drag it to the desktop, and a clipping file is created. Drag and drop work as expected, too.

Choose Edit > Find > Find. In the Find dialog, enter a word that appears in the text and click Next. Find works as expected. Choose Edit > Spelling, then

Grammar > Show Spelling and Grammar. Spell checking works as expected. Test the Edit menu's Substitutions, Transformations and Speech menu items. They all work as expected.

A few things aren't yet working. For example, drag the divider in the diary window up to expose the lower pane of the split view. You can click in it and type to enter text, but a little experimentation quickly establishes that this is not another view into the same text store (**Figure 3.11**). Choose File > Save As. The usual Save panel opens, but when you try to save the document, an error alert reports that the document could not be saved. Clearly, not everything comes for free in Cocoa, but an awful lot does. You'll fill these gaps in the next two steps.

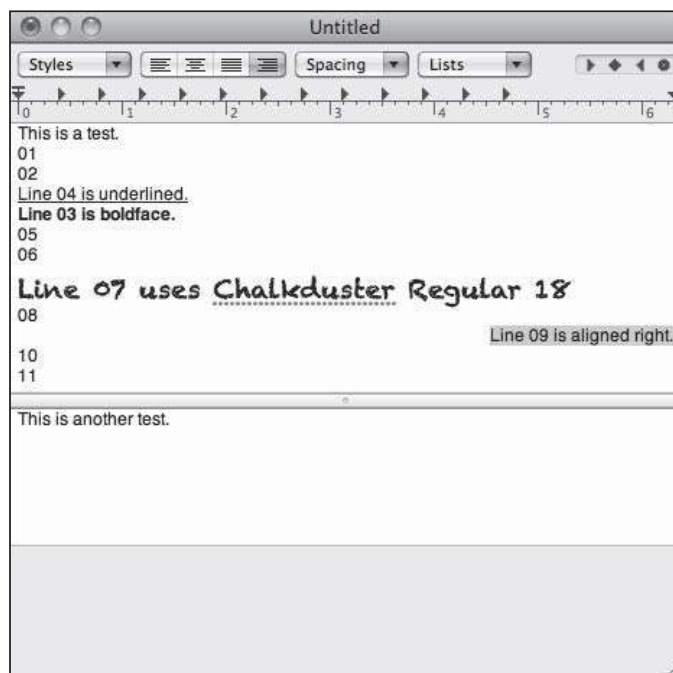


FIGURE 3.11 The Chef's Diary window with formatted text in one pane.

Step 7: Read and Write the Diary Document's Text Data

The role of the Chef's Diary window in the Vermont Recipes application is to save the user's culinary experiences for posterity. This necessarily implies that the text that the user types will not disappear forever when it scrolls off the top of the window. Instead, the user can scroll back up and reread it. It also implies that the text will not disappear when the user turns off the computer. Instead, the user can retrieve it for reading after turning the computer back on.

The *model* in the MVC design pattern handles both of these requirements. In this step, you learn how the Cocoa text system handles the persistence of text data in an application, both in the short term and in the long term. It is a good introduction

to the model part of the MVC design pattern, because Cocoa creates the model for you. There is almost nothing required of you.

In this step, you will learn how to take advantage of the Cocoa text system to handle all of the data persistence needs of the Diary window. Instead of requiring the user to commit pending edits explicitly, the application will update the data in the model continuously as the user types, cuts, pastes, drags, and sets styles and fonts. The user's experience will be seamless, as it should be in any word processor. The user will be able to edit for a while, and then choose File > Save to save recent progress, edit some more, save again, and so on, all without having to interrupt the workflow to commit the data by pressing Enter or clicking a Done Editing button before saving.

This step will not go into much detail about the Cocoa text system. The text system is deep, complex, flexible, and enormously powerful, yet at the same time it can be used in simple ways to perform simple tasks. Here, you will use only those features that are needed for a simple diary having no need for complex layouts. Read the “Cocoa Text System” sidebar for an introduction.

The Cocoa Text System

The Cocoa text system provides the model, the view, and the controller in one big package of classes. It is only a slight exaggeration to say that you have practically nothing left to do except to tell the text system how many layout managers, text containers, and text views to associate with any particular text storage object. Text systems are among the most complex topics in application programming, but Cocoa has relieved you of almost all the tedium.

Everything works through delegate methods by design, and you almost never have to subclass any of the text system's classes.

- `NSTextStorage` is a subclass of `NSMutableAttributedString`, so it consists mostly of the text and associated formatting attributes. As such, it is considered the MVC model of the text system. It contains a list of its layout managers and other information in addition to the mutable attributed string.
- `NSLayoutManager` controls the layout of glyphs in the text container, described next. As such, it is considered an MVC controller object. It controls word wrap within the text container and many similar behaviors.
- `NSTextContainer` provides the onscreen space in which the text will be laid out. It is rectangular, but you can subclass it to provide different shapes. It is also considered an MVC controller object.
- `NSTextView` is the MVC view object in which the user types and reads the text, makes selections, and performs editing operations.

(continues on next page)

The Cocoa Text System *(continued)*

When writing a Cocoa program, you normally come at it from one or the other end of this list. If you're using Interface Builder, you drag a text view from the Library window into a window and normally don't have to think about the other components. Interface Builder puts together the network of classes underlying the text view for you, and it all just works. You usually keep a reference to the text view and use it in your code as needed.

If you're using Xcode to construct a text view programmatically, you usually start by instantiating and initializing a text storage object, followed by its layout manager, followed by its text container, and followed finally by its text view, linking all of these together. You often keep a reference to the text storage object and use it to get at everything else. In fact, the other components are usually released immediately after you instantiate them and add them to the text storage object, which thereafter owns them.

You would usually use the latter approach if you needed to construct anything more complicated than the simple stand-alone text view you can create in Interface Builder. For example, you can programmatically create combinations of the main text system components that allow a single text storage to flow automatically across page breaks or column boundaries or to appear in multiple panes of a split view window. One text storage can have multiple text layout managers and one layout manager can have multiple text containers, for example, each resulting in a different combination of features and interrelationships.

Here, you use an `NSTextStorage` object—essentially, a string with formatting attributes attached—to hold the diary's text. `NSTextStorage` inherits from `NSAttributedString` through `NSMutableAttributedString`. It not only contains the formatted string, but it also implements methods to link to the other objects that make up the Cocoa text system, ultimately including the text view. The text view that is instantiated when you open the diary window has its own text storage object and related objects, which handle almost all of the coordination between the user's typing and the text. All you have to do is declare a couple of instance variables, implement a delegate method or two, and implement the required methods to store and retrieve the data on disk.

There is no need to create a separate model class to isolate the code relating to the diary document's data from the control functions in the `DiaryDocument` class you already created. The Cocoa text system provides the model for you in the form of its `NSTextStorage` class. All you have to do is write a little code in the `DiaryDocument` class to communicate with the `NSTextStorage` object.

1. Open the `DiaryDocument.h` header file and declare the `diaryDocTextStorage` instance variable between the braces of the `@interface` declaration:

```
NSTextStorage *diaryDocTextStorage;
```

2. Declare accessor methods in the header to get and set the value held in the `diaryDocTextStorage` instance variable:

```
- (void)setDiaryDocTextStorage:(NSTextStorage *)textStorage;  
- (NSTextStorage *)diaryDocTextStorage;
```

3. Open the `DiaryDocument.m` implementation file and implement the accessor methods:

```
- (void)setDiaryDocTextStorage:(NSTextStorage *)textStorage {  
    if (diaryDocTextStorage != textStorage) {  
        [diaryDocTextStorage release];  
        diaryDocTextStorage = [textStorage retain];  
    }  
}  
  
- (NSTextStorage *)diaryDocTextStorage {  
    return [[diaryDocTextStorage retain] autorelease];  
}
```

Read the “Instance Variables and Accessor Methods” sidebar for more information.

Instance Variables and Accessor Methods

One way the application could set the value of the `diaryDocTextStorage` instance variable would be to assign it directly to the instance variable itself.

However, to isolate your application’s underlying implementation as much as possible from the code that gets and sets its value, you should instead add accessor methods. By doing this, you give yourself the freedom to change the innards of the application without altering the declaration of the accessor methods in the header file. In this fashion, you avoid forcing clients that use the header file to recompile when you make changes to the underlying implementation details. You would have to revise the implementation of the accessor methods in the implementation file only to accommodate the underlying change. The use of accessor methods is a commonplace of Cocoa development for this reason, among others.

(continues on next page)

Instance Variables and Accessor Methods *(continued)*

This is true to such an extent that Cocoa is actually written to look for accessor methods conforming to certain conventions. To take full advantage of the benefits conferred by Cocoa, you should always honor these conventions. In summary, an accessor method that gets the value of an instance variable should have the same name as the instance variable and simply return its value. An accessor method that sets the value of an instance variable should have the same name as the instance variable, but with its first letter capitalized, preceded by `set` as a prefix. Thus, the accessor methods for an instance variable named `myWidth` would be `myWidth` and `setMyWidth:`. Obeying this convention is even more important for your application's data values than it is for its controls and other view objects.

You can warn clients of your class to rely only on the accessor methods by declaring the associated instance variable `private`. Still, it is a peculiarity of Objective-C that instance variables are visible and accessible in the header file despite the `@private` compiler directive. If you want to get really fancy, you can keep clients completely ignorant of your methods by declaring them in a separate category that you declare in your implementation file, but this is an advanced technique, and it is not available for instance variables.

Most instance variables that refer to a control as opposed to a data value have only a `get` accessor. You don't need a `set` accessor because you are not going to create multiple copies of a control in code. You do want a `get` accessor, because you will access the control very frequently to obtain a reference to its visible state so that you can retrieve or change its appearance. You could simply refer to them through their instance variables. You will take a somewhat pedantic approach here and declare accessor methods for them anyway, at the expense of adding to the heft of the application's source code.

The specific code used in the accessor methods you write in this recipe involves memory management considerations that you will address later.

These are standard getter and setter accessor methods, but this setter method embodies an important subtlety. In its last statement, it retains the `textStorage` parameter value instead of copying it. Books and guidance documents relating to Cocoa advise you that, most of the time, you should **copy** data-bearing objects such as strings and **retain** only those objects that can safely share the same memory. Copying reproduces the bits of the original object in another location in memory, so that later changes to one of the objects will not cause changes to the other. Retaining simply increments the change count of the object, and changes later made to that object are reflected in all references to it.

In the Diary document, the text might become very lengthy, and copy operations might therefore consume a noticeable amount of time. You therefore use **retain**, not **copy**, and any text storage operation using this setter method will be quite fast no matter how much text is involved.

You will have to remain alert as you write the rest of the code, however, to make sure you don't inadvertently violate the plan to use a single text storage object for all the document's text operations. Recall that, in a typical document-based application, the *controller* in the MVC design pattern is broken down into two controllers that work closely with one another. One, a subclass of `NSDocument`, is thought of as a *model-controller* and specializes in managing the model side of communications between the model and the view. The other, a subclass of `NSWindowController`, is considered a *view-controller* and specializes in managing the view side of communications between the model and the view. It is important to keep these roles clear in your mind as you continue to work with the text view in this step.

In `DiaryDocument` and `DiaryWindowController`, it is easy to confuse these roles. The model—namely, an instance of the `NSTextStorage` class—is part of the Cocoa text system. The design of the Cocoa text system tends to make one associate the text storage object with the text view, because instantiating a text view automatically sets up its associated text storage object. Nevertheless, the text storage object actually exists in its own right, independently of the text view, and using the `DiaryDocument` accessor methods you just wrote lets the document refer to them on equal standing with the text view controlled by `DiaryWindowController`.

In the accessor methods you just wrote, you see a model-controller in action. Conceptually, the `-setDiaryDocTextStorage:` setter method receives a text storage object from somebody who has it and puts it into what is, conceptually, the model, pointed to by the `diaryDocTextStorage` instance variable declared in the `DiaryDocument` class. In the other direction, the `-diaryDocTextStorage` getter method obtains the value of the `diaryDocTextStorage` instance variable from the model and provides it to somebody who wants it. In both cases, as you will soon see, the somebody who is the source of data to be placed in the model and the destination of data to be retrieved from the model is the disk store accessed when the document is saved and when it is opened.

Behind the scenes, the view accesses the same text storage object, so changes made when the user types in the text view are simultaneously available to the document without the need to move any bits.

4. Now that you have written the getter and setter methods on the model-controller side in the `DiaryDocument` class, you must write methods on the view-controller side in the `DiaryWindowController` class so that the document can get what the user has typed and store it on disk, and send to the user what

it has retrieved from disk. Note that you do not need a setter accessor method, because the text view is set up in Interface Builder and you will never need to set it again.

First, you need a way to access the text view in order to get its text storage object, just as you have already created a way to access the text storage object in the document. You did not create an outlet to the split view or any of its parts when you created the view in Step 4, so you'll have to do it now.

The task is complicated by the fact that the split view contains two text views. The application specification calls for both panes of the split view to display the contents of the same diary, so work with the top pane's text view for now. You'll fix up the bottom pane in Step 8.

Open the `DiaryWindowController.h` header file and declare the `diaryView` instance variable between the curly braces of the `@interface` directive:

```
IBOutlet NSTextView *diaryView;
```

5. Also declare a getter accessor method for the instance variable:

```
- (NSTextView *)diaryView;
```

6. Open the `DiaryWindowController.m` implementation file and implement the accessor method using the same pattern you used for the getter `diaryDocTextStorage` accessor method:

```
- (NSTextView *)diaryView {  
    return [[diaryView retain] autorelease];  
}
```

7. Connect the outlet to the text view in the top pane of the split view in the `DiaryWindow` nib file. To do this, first select the File's Owner proxy in the nib file. Then drag from the marker beside the `diaryView` outlet in the `Diary Window Controller Connections` inspector to the text view in the top pane of the scrolling split view in the diary window. It can be difficult to connect the outlet to the text view because the scroll view and the split view are in the way. The text view is available in the uppermost area of the top pane of the split view in the diary window to make it easy to drop connections onto it.
8. With that out of the way, you're ready to get to the heart of the matter.

Consider the fact that a user opens a document window in two fundamentally different situations: (1) when creating a new, empty document in anticipation of typing new text into it and saving it to disk; and (2) when opening a document from disk in anticipation of reading existing text and, perhaps, editing it and saving it back to disk. Your application can respond to both of these situations by implementing one of two methods that are declared in the Cocoa frameworks for this purpose. You need to override one of them in `DiaryWindowController` in order to set up the diary window's text views.

Traditionally, developers have done this by overriding the `-awakeFromNib` method, which is called on every object that is stored in the nib file as soon as the nib file has finished loading and connecting all of its objects. However, Apple engineers informally encourage developers to override the `-windowDidLoad` method, instead, when implementing a window controller subclass. Every window controller calls it just after calling `-awakeFromNib`, but no other kind of object calls it, so you don't have to worry about its being called multiple times when a single nib file loads. Cocoa automatically calls your implementation of either or both of these methods, if you override them, every time the user loads the DiaryWindow nib file by opening the window. You should follow the engineers' advice and override `-windowDidLoad`.

Still in the `DiaryWindowController.m` implementation file, implement the `-windowDidLoad` method:

```
- (void)windowDidLoad {
    NSTextStorage *docTextStorage = [[self document] diaryDocTextStorage];
    if (docTextStorage == nil) {
        [[self document] setDiaryDocTextStorage:
         [[self diaryView] textStorage]];
    } else {
        [[[self diaryView] layoutManager]
         replaceTextStorage:docTextStorage];
    }
}
```

The `-windowDidLoad` method is declared and implemented in `NSWindowController` and called automatically at the appropriate time, but its implementation does nothing. Like many methods implemented in the Cocoa frameworks, which either do nothing or perform some simple default operation, `-windowDidLoad` is intended to be overridden by subclasses. Some Cocoa classes do nothing but declare methods and implement them as stubs. Such a class is known as an *abstract class*, and you must implement a *concrete subclass* to make things work. Other classes, like `NSWindowController`, are themselves concrete classes, but they contain some abstract methods that you may, or in some cases must, override.

In your override of `-windowDidLoad`, you start by assigning the document's text storage object to the `docTextStorage` local variable. This is simply a coding convenience, so that you can refer to the document's text storage object in later statements instead of typing out the longer reference.

In the `if` test, you test the `docTextStorage` value to see whether it is `nil`. It is `nil` only when the user opens a new, empty window whose associated document has not yet read any text from disk. If it is not `nil`, then you know that the `-windowDidLoad` method has been called because the user just opened an

existing file on disk and read its text into the document's text storage object. You'll see how the document reads the text from disk in a moment.

If the test shows that the user has just opened a new, empty window, you assume that the user is about to start typing, thus adding content to the text storage object that is associated with the new text view. The document associated with the window will therefore soon need a reference to the text storage in order to write it to disk. You simply set the document's text storage to the diary view's text storage.

In the `else` branch of the test, the user has just read the text from disk into the document's text storage object. You therefore replace the empty text storage object in the text view's layout manager with the document's text storage object held in the `docTextStorage` local variable. You will see shortly how the document's data retrieval mechanism reads the text from disk and places it in the document's `docTextStorage` object, before the empty window is loaded from the nib file and opened.

`NSTextView` doesn't itself implement `-setTextStorage:` or `-replaceTextStorage:` methods, so you could not simply call `[[self diaryView] replaceTextStorage:docTextStorage]`. Instead, the Cocoa text system implements these methods in `NSLayoutManager`, which lays out the text correctly in the view according to its size, shape, and other layout options.

After the text view's text storage object is replaced with the document's text storage object just read from disk, the document's text storage object and the window controller's text storage object are one and the same object. This is again consistent with your goal of having a single text storage object that always contains the text most recently typed by the user or read from disk.

Take a moment to understand how this works, looking backward in time from the `-windowDidLoad` method, to see how it was called.

- Cocoa is wired internally to call your override of `-windowDidLoad` whenever the diary window nib file is loaded.
- You arranged to make the document load the nib file when the user opens a saved diary document, by writing `DiaryWindowController`'s `-init` method to load the nib file named `DiaryWindow`.
- You wrote `DiaryDocument`'s `-makeWindowControllers` method to create the window controller and call its `-init` method.
- `NSDocumentController` lies behind all this, fulfilling its role in Cocoa's document-based application architecture by calling your `-makeWindowControllers` method from its `-openDocument:` method.

- The MainMenu nib file created by the document-based application template includes an Open menu item in the File menu, which is connected to the First Responder and calls the document controller's `-openDocument:` action method
9. The last step is to implement methods to write the text in the document's text storage object to disk and read it back from disk. This is the most important role of the DiaryDocument subclass of NSDocument.

Recall from Step 1 that the NSDocument template came with three stub method implementations, one of which you promptly deleted. Two of them, NSDocument's `-dataOfType:error:` and `-readFromData:ofType:error:` methods, have default implementations in NSDocument, but they are meant to be overridden by subclasses in most applications. Override them now by filling in the stubs inserted in the DiaryDocument.m implementation file by the template.

When you define a custom format for a document's data, you typically store and retrieve the data by using Cocoa's NSKeyedArchiver and NSKeyedUnarchiver classes in your overrides of these two methods. Those classes know how to encode and decode information in any object that conforms to the NSCodering protocol to and from an NSData object. An NSData object is a serial collection of bits, and it is therefore suitable for storage on disk. Every data object in the Cocoa frameworks that is intended to be stored on disk using NSKeyedArchiver adopts the NSCodering protocol and implements its methods.

Cocoa provides other mechanisms for storage of specialized forms of data, however, including methods to store and retrieve RTF text. When you use the RTF methods, any application that supports RTF text, such as TextEdit, can read and edit the RTF files that Vermont Recipes creates to save the diary.

In the DiaryDocument.m implementation file, erase the comments inserted by the template explaining what to do with the `-dataOfType:error:` and `-readFromData:ofType:error:` methods. Then implement them as follows:

```
- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError {
    NSRange range = NSMakeRange(0, [[self diaryDocTextStorage] length]);
    NSData *data = [[self diaryDocTextStorage] RTFFromRange:range
                    documentAttributes:nil];
    if ((data == nil) && (outError != NULL)) {
        *outError = [NSError errorWithDomain:NSCocoaErrorDomain
        code:NSFileWriteUnknownError userInfo:nil];
    }
    return data;
}
```

```

- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName
    error:(NSError **)outError {
    NSTextStorage *textStorage = [[NSTextStorage alloc]
        initWithRTF:data documentAttributes:NULL];
    if ((textStorage == nil) && (outError != NULL)) {
        *outError = [NSError errorWithDomain:NSCocoaErrorDomain
            code:NSFileReadUnknownError userInfo:nil];
        return NO;
    }
    [self setDiaryDocTextStorage:textStorage];
    [textStorage release];
    return YES;
}

```

Cocoa automatically calls your override of the `-dataOfType:error:` method when the user chooses File > Save As or performs other standard actions to save the diary document.

In the first statement in the `-dataOfType:error:` method, you call one of Cocoa's many useful functions, `NSMakeRange()`, to create a value of type `NSRange` and assign it to the `range` local variable. The `NSRange` type is not a Cocoa object but a C *struct* consisting of two values, in this case the index of the first character of the text returned by the `-diaryDocTextStorage` method you wrote earlier, followed by the number of characters in the range. You want to store all of the characters, of course, so you create a range that starts with the first character, at index 0, and encompasses the entire `length` of the string contained in `diaryDocTextStorage`. `NSAttributedString`'s `-length` method counts the number of characters in its unformatted string.

Remember that `NSTextStorage` is a subclass of `NSMutableAttributedString` and, in turn, of `NSAttributedString`. It is therefore correct to instantiate an `NSTextStorage` object and call a method it inherits from `NSAttributedString`.

In the second statement, you call `NSAttributedString`'s `-RTFFromRange:documentAttributes:` method, which returns an `NSData` object, and you return it as the method's result if it is not `nil`. `NSTextStorage` does not implement an `-attributedString` method because it *is* an attributed string. Cocoa takes the `NSData` result and saves it to disk without further effort on your part.

In the third statement, you return an `NSError` object by reference in the last parameter of the `-dataOfType:error:` method if `-RTFFromRange:documentAttributes:` returns `nil` and the `outError` argument is not `NULL`. If the attempt to write the document's contents fails, Cocoa will present an error alert. The `NSError` mechanism employed by Cocoa will be explained in detail in Recipe 6.

There are two features of the `-dataOfType:error:` method that you won't make use of for now. One is the `ofType` parameter. When Cocoa calls the method, it places a string value in this parameter, representing the document's type obtained from the Info.plist document type entries you created in Step 6. For a diary document, this is `com.quecheesoftware.vermontrecipes.diary`. Your implementation of the method can use this value or ignore it, depending on whether you need to make decisions based on the type. Finally, Cocoa uses the `documentAttributes` parameter of the `-initWithRTF:documentAttributes:` method to store document attributes for documents that have them. Pass `nil`, because this document has none.

Cocoa automatically calls your override of the `-readFromData:ofType:error:` method when the user chooses File > Open or takes other standard actions to open a diary document from disk.

The first statement allocates memory for an `NSTextStorage` object and initializes it with the `-initWithRTF:documentAttributes:` method that it inherits from `NSAttributedString`. It then assigns the result to the `textStorage` local variable. The result is not `nil` if no error occurred. In that case, the method assigns the value of `textStorage` to the document's `diaryDocTextStorage`, replacing its previous value. The memory set aside when you instantiated the `textStorage` object is then released, because there is no need for the temporary local variable after its value has been assigned to the document's `diaryDocTextStorage`. The method then returns `YES` to signal success.

If an error did occur, the `textStorage` local variable is `nil`. In that case, the method returns an `NSError` object by reference in the third parameter of the `-readFromData:ofType:error:` method and returns `NO`.

Cocoa's document-based application mechanism reads the text from disk before the document window and its nib file have been loaded. For that reason, the document must hang on to the text in its `diaryDocTextStorage` instance variable until the window is ready. Recall that the document-based application mechanism calls your implementation of the window controller's `-windowDidLoad` method, after the document has read the text from disk, and sets up the text view to use the document's text storage object.

10. You've done a lot of work in this step, and being able to store and retrieve the diary is crucial to the application, so test it now.

Build and run the application. Choose File > New Chef's Diary to open the diary window, and then type a word or two in the top pane. Notice that the window's close button now indicates that the document contains unsaved changes. Next, choose File > Save As or File > Save, or simply click the win-

dow's close button. If you click the close button, a standard sheet appears, asking whether you want to save the document or cancel the close operation. Choose Save. A standard Save panel appears. In the Save As field, the default name *Untitled* appears, selected so that you can easily change it, followed by the *vrdiary* file extension you specified in the Info.plist file. Enter a name, navigate to the desktop, deselect the "Hide extension" checkbox, and click Save. After a moment, the file's icon appears on the desktop with the name you assigned it and the *vrdiary* file extension. Because you haven't yet provided an icon for this document type, Cocoa provides the generic document icon. If you have turned on the "Show icon preview" setting in the Finder's view options, the file extension *vrdiary* appears on the icon itself.

Close the diary window, and then choose File > Open. In the standard Open panel, navigate to the desktop, select the new file, and click Open. The only files that are enabled so that you can choose them are files with the *vrdiary* extension and any generic RTF files that may already exist in the folder. Alternatively, drag the file's icon from the desktop onto the Vermont Recipes application icon in the Dock or double-click the file's icon. The diary window opens, displaying the text you saved.

You can even use TextEdit or any other RTF-capable application to read and edit the diary document. Open the Applications folder and drop the new diary document on TextEdit's file icon. The document opens in TextEdit. Make some changes and save them, and then drag the document's icon onto the Vermont Recipes icon in the Dock. The file opens in Vermont Recipes' diary window, and the changes you made in TextEdit are visible.

Step 8: Configure the Split View Diary Window

You have one remaining task in this recipe: to make the second pane of the split view in the diary window work. Recall that the application specification calls for two panes providing views into the same document, so that the user can read one part of the document while typing in another part, just as you often do in Xcode text editing windows.

It's a straightforward task. In Step 7, you learned that you can easily replace a text view's text storage object within its layout manager. Do the same thing with the two text views in the diary window, and both views then display text from the same text storage object in different layout managers.

1. Start by setting up an instance variable for the second text view in the `DiaryWindowController.h` header file and declaring two accessor methods for it. Between the two braces of the `@interface` directive, declare the outlet immediately below the `diaryView` outlet:

```
IBOutlet NSTextView *otherDiaryView;
```

Below the accessor for `diaryView`, declare the getter accessor method for `otherDiaryView`:

```
- (NSTextView *)otherDiaryView;
```

2. In the `DiaryWindowController.m` implementation file, implement it:

```
- (NSTextView *)otherDiaryView {  
    return [[otherDiaryView retain] autorelease];  
}
```

3. Return to the `-windowDidLoad` override method you implemented in the `DiaryWindowController.m` implementation file. Add this statement at the end, following the code you wrote to give the document and the upper diary view references to the same text storage object:

```
[[[self otherDiaryView] layoutManager]  
    replaceTextStorage:[self diaryView] textStorage];
```

Now a single text storage object is used by the document, the upper text view and the lower text view.

4. Connect the new `otherDiaryView` outlet to the bottom pane in Interface Builder in the same way you connected the `diaryView` outlet to the top pane in Step 7.
5. It is a useful exercise to build and run the application now so that you can test the two panes of the split view. Open an empty diary window, drag the divider about halfway up so that you can see both panes, and start typing in the upper pane. No matter how fast you type, you see the same text appear simultaneously in the lower pane. This is good news, but, unfortunately, you aren't yet home free. Click in the lower pane, press the Return key to start a new line, and type something else. There is a noticeable delay before your typing appears in the upper pane.

The problem is that the upper pane wasn't immediately aware that you had made changes to the shared text storage object. In Cocoa, you often encounter situations like this. To deal with them, there is a standard mechanism to inform the offending view that it's time to wake up. Many views implement methods whose names signal that they need to be displayed. `NSTextView` implements just such a method, `-setNeedsDisplayInRect:avoidAdditionalLayout:.` This method, combined with judicious use of an appropriate delegate method, solves the problem.

NSTextView provides many delegate methods that your applications can use to react to the user's actions. One that is commonly used and is perfect for this situation is `-textDidChange:`. If the delegate of each of the diary windows' text views implements this delegate method, the text view calls it repeatedly as the user types or changes the formatting attributes of the text.

First, set the `DiaryWindowController` object as each text view's delegate. Perhaps the easiest way to do this is to set the nib file's document window to outline view and expand the Window object until you see the two text views in the custom views of the split view. Then Control-click the top text view to open its HUD, and drag from its delegate outlet to the File's Owner. Repeat the process on the bottom text view.

Then add this delegate method to the `DiaryWindowController.m` implementation file:

```
- (void)textDidChange:(NSNotification *)notification {
    if ([notification object] == [self diaryView]) {
        [[self otherDiaryView]
         setNeedsDisplayInRect:[self otherDiaryView] visibleRect]
        avoidAdditionalLayout:NO];
    } else {
        [[self diaryView]
         setNeedsDisplayInRect:[self diaryView] visibleRect]
        avoidAdditionalLayout:NO];
    }
}
```

Many delegate methods have a single parameter, a notification object. The notification object always has an `object`, and it frequently has a `userInfo` object as well. Both contain information that you can use in your implementation of the delegate method to help it do its job. This delegate method only has an `object`, which the documentation discloses is a reference to the text view that triggered the delegate method when the user made some changes to it. You use this parameter to determine whether it was the upper or the lower pane that the user was editing, and you tell the other text view that it needs to display itself immediately.

In both cases, the rectangular area to be updated is the `visibleRect` of the text view. Text views can be very large, and you don't want to slow down the application any more than you have to in order to get its formatting right. For most text views, this means the part of the text view extending from its beginning to the part appearing at the bottom of the text view based on the current setting of the scroller.

You pass `NO` to the `avoidAdditionalLayout` parameter, because you do want the text in both panes to be properly formatted and laid out.

Step 9: Build and Run the Application

As you do at the end of every recipe, build and run the application to test what you've created.

You already know that you can open a new diary window, change it, save it, close it, open it, change it again, save it again, or even save another version of it. You should try everything else you can normally do with documents, to satisfy yourself that it is working correctly. Choose File > Revert to Saved after making some changes to a saved document. Use the Open Recent menu item in the File menu. Use the Undo and Redo menu items in the Edit menu. Almost everything works.

Also try different combinations of typing and formatting in the two panes of the window, to be sure that text and other changes appear simultaneously in both (**Figure 3.12**).

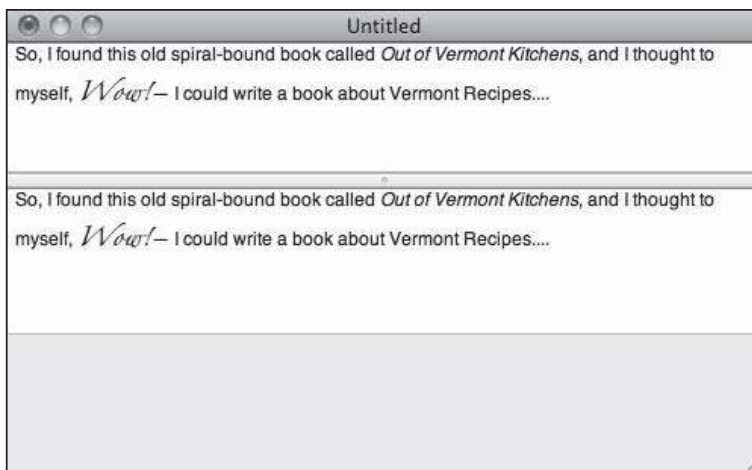


FIGURE 3.12 The Chef's Diary window with identical text in both panes.

Finally, experiment with a variety of RTF files, whether created in Vermont Recipes or any other RTF-capable application.

Step 10: Save and Archive the Project

Now that you're done with this step and satisfied that everything is working that should be working at this point, delete all the snapshots you've accumulated. You're about to archive the project, and there is no need to leave your hard drive cluttered with older snapshots.

Quit the running application, close the Xcode project window, and save if asked to do so. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 3.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 4.

Conclusion

You have taken the first important step toward a fully functional Chef's Diary. In the next two recipes, you will add several controls to the bottom of the diary window, as well as a menu and a few menu items to give the user alternative ways to exercise the controls. You will also, of course, hook up the new controls and menu items to make everything work. You will be amazed at how powerful yet easy to use the Chef's Diary will be.

You aren't likely to create a text editor along the lines of the Chef's Diary in every application you write, but having a basic understanding of the Cocoa text system will prove surprisingly useful. In addition, the techniques you used to save and retrieve the contents of the Chef's Diary have given you a good idea of how to go about saving and retrieving any document's data.

Documentation

Recipe 3 covers a range of topics relating to creation of documents and disk storage. In exploring these topics, you learned about several related technologies, including Uniform Type Identifiers and text views. After you work through this and other recipes, it is a good idea to read Apple's documentation on the same topics.

Class Reference and Protocol Documents

Every class in the Cocoa AppKit and Foundation has its own *Class Reference* document, where you find a brief description of the class and what it does followed by an organized table of contents labeled Tasks. Sometimes the general description is long and detailed, containing valuable information you won't find anywhere else. It also includes inheritance information and other information about the class. Each method in the class is described separately with a brief description of what it does and how to use it, along with descriptions of its parameters, return value, cross-references to related methods, references to general documentation and sample code, and the name of the header file where it is declared. Starting with Snow Leopard, you should also look at the separate *Protocol Reference* document for delegate methods of any class that declares delegate methods.

Take careful note of the inheritance information about every class. The Cocoa frameworks make good use of the object-oriented capabilities of Objective-C. If you forget that every class above the root class inherits methods from its superclass, you will overlook major capabilities of the classes you use.

You should read the Class Reference document for every class you encounter. It's the best way to become familiar with the extent of the Cocoa frameworks.

(continues on next page)

You should also look at the header file for every class. They sometimes contain detailed comments about usage of the class and its methods that you won't find in the documentation. On occasion, you'll even find methods in the header that aren't documented at all in the *Class Reference* document.

As to Recipe 3, you should read the following, listed in the order in which you encountered the classes:

NSDocument Class Reference

NSWindowController Class Reference

NSObject Class Reference

NSTextView Class Reference

NSTextDelegate Protocol Reference

NSDocumentController Class Reference

NSString Class Reference

NSAttributedString Class Reference

NSAttributedString Application Kit Additions Reference

NSMutableAttributedString Class Reference

General Documentation

Apple publishes an extensive set of programming guides and overview documents explaining how to accomplish particular tasks. Some of them are introductory in nature, whereas others go into detail about all the options available.

You should read the following documents about topics covered in Recipe 3:

Document-Based Applications Overview

Text System Overview

Text System Storage Layer Overview

Text System User Interface Layer Programming Guide for Cocoa

Runtime Configuration Guidelines

Uniform Type Identifiers Overview

Simplifying Data Handling with Uniform Type Identifiers

Release Notes

You should read the Cocoa AppKit and Foundation release notes for every major release of Mac OS X, especially recent releases. They often contain detailed information about usage of classes and specific methods that never finds its way into other documentation.

With respect to technologies covered in Recipe 3, the Leopard and Snow Leopard Application Kit release notes contain a wealth of information about Uniform Type Identifiers.

Add Controls to the Document Window

In Recipe 3, you created a powerful text editor, and you arranged to save its contents to disk and to read them back so that the user can maintain a diary recording notable culinary events. You refined the original Vermont Recipes application specification to provide for several controls at the bottom of the diary window to help update and use the diary. You add the controls and wire them up in this recipe.

Controls are a commonplace of every application. They range from the simplest push button to complex devices like the date picker. The basic techniques you use to create them and make them work are similar for all of them.

In this recipe, you start by creating two simple push buttons to insert the title of a new diary entry and to add tags to an existing entry. Next, you create four navigation buttons, with graphics, to enable the user to scroll to the previous or next diary entry and to the first and last entries. Finally, you create two complex controls, a date picker and a search field, for more sophisticated navigation within the diary.

In addition to learning how to create controls, you learn in this recipe about programmatic manipulation of text in the Cocoa text system.

HIGHLIGHTS

Creating simple push buttons

Setting tab order in a window (the key view loop)

Writing action methods and using the sender parameter

Using the NSLog() function for debugging

Using format strings

Displaying Unicode characters

Manipulating text in the Cocoa text system

Controlling undo and redo in the Cocoa text system

Validating (enabling and disabling) buttons and other controls

Using Objective-C selectors

Creating and using Objective-C formal protocols

Creating push buttons with images

Creating a date picker

Using dates in Cocoa

Creating a search field

Step 1: Add Controls to the Diary Window

The detailed specification for the Chef's Diary at the beginning of Recipe 3 calls for several controls in the space at the bottom of the window. You start in this step by using Interface Builder to add the controls to the window (**Figure 4.1**). In subsequent steps, you will hook each of them up in turn to perform a specific task.



FIGURE 4.1
The Vermont Recipes
Chef's Diary window.

1. Start by opening the Vermont Recipes 2.0.0 folder in which you left the working Vermont Recipes project folder at the end of Recipe 3, leaving the compressed project folder you archived at that time where it is. Open the Vermont Recipes.xcodeproj file in the working project folder to launch Xcode and open the project window.

Increment the CFBundleVersion value by selecting the Vermont Recipes target in the Targets group, opening its information window, and selecting the Properties tab. Change the value in the Version field from 3 to 4, and save and close the information window. When you open the About window after building and running the application at the end of this recipe, you will see the application's version displayed as 2.0.0 (4).

2. Open the DiaryWindow nib file in Interface Builder.
3. In the Library window, navigate to Library > Cocoa > Views & Cells > Buttons, and drag two Push Buttons to the empty space at the bottom of the diary window. Position them one above the other toward the left side of the window,

using the guides that temporarily appear to help you put them in the right place. These guides conform to the requirements of Apple's *Human Interface Guidelines*, making it easy for you to build a user interface that meets Mac users' expectations. The bottom button should snap to the guides defining the margins at the left and bottom edges of the window. The top button should snap to the guides for defining the margin at the left edge of the window and the spacing above the bottom button. If you need more room, Command-drag the bottom of the split view higher.

4. Double-click the top button to select its title for editing, or click in the Title field in the Button Cell Attributes inspector, and enter **Add Entry**.
5. Using the same technique, rename the bottom button **Add Tag**.
6. If you look closely, you notice that the title of the top button was a little too long to fit in the default width of the button, and Interface Builder automatically widened it to make room as you typed. Select the bottom button, and drag its left and right edges and move it as needed until guides appear, showing you that it is lined up with the left and right edges of the top button.

Another way to make the buttons the same size is to use the Button Size inspector. Before you resized the bottom button, the W (for *width*) field in its Button Size inspector and the same field in the Size inspector for the top button indicate different widths. You could simply have changed the W field in the Button Size inspector for the bottom button to match that for the top button. The button would have grown wider when you committed the change. You might still have had to move it to line up with the guides.

7. Before fixing the two new buttons' autosizing behavior, have a little fun with their default behavior. Choose File > Simulate Interface and resize the window. You see that their locations remain fixed relative to the upper-left corner of the window, creating the comical impression that they are moving up into the split view or disappearing off the bottom edge of the window.

Quit the Cocoa Simulator. To fix the problem, select the top button. In the Autosizing section of the Button Size inspector, disable the top strut and enable the bottom strut, leaving the left strut enabled. Do the same with the bottom button. This is standard practice with controls that are located in the bottom-left quadrant of a resizable window.

Now run the Cocoa Simulator again, and the buttons behave properly, remaining locked in place relative to the lower-left corner of the window.

8. Go back to the Library window and navigate to Library > Cocoa > Views & Cells > Inputs & Values. Scroll down to the Date Picker and drag it to the upper-right corner of the empty space at the bottom of the Chef's Diary window.

9. In the Date Picker Attributes inspector, make sure the “Month, Day and Year” and the “Hour, Minute, and Second” radio buttons are selected, and leave the other settings as you find them. Once you’ve finished this recipe, each entry in the diary will be marked by a date-time heading, down to the second, and the date picker will automatically display the date and time of the current entry. Displaying the seconds allows users to make multiple entries less than a minute apart.
10. Select the date picker in the window. You don’t see the hour, minute, and second entries because by default the control is sized too small to reveal them. Choose Layout > Size to Fit, and the control widens to show both the date and the time elements. The button now runs off the edge of the window, so reposition it with the help of the guides.
11. In the Date Picker Size inspector, disable the left and top struts and enable the right and bottom struts to lock the control to the lower-right corner of the window.
12. Go back to the Library window, and still in the Inputs & Values section, drag a Search Field into the diary window and drop it below the date picker, using the guides to position it properly in the corner. The heights of the Add Tag push button and the search field are different. I normally use the guides to line up the center or the text baselines in this situation. This can be done by selecting the Add Entry button and the search field together and choosing Layout > Alignment > Align Horizontal Centers or Align Baselines.
13. Drag the search field’s left edge to align with the left edge of the date picker, so that they are the same length.
14. In the Search Field Size inspector, set the struts the same as you did for the date picker.
15. Return to the Library window, and in Library > Cocoa > Views & Cells > Buttons, select a Square Button. Drag it into the diary window and drop it immediately to the left of the date picker.
16. In the Button Size inspector, note that the square button is 48 pixels wide by 48 pixels high. Enter **24** in both the W and H fields. After you commit each entry by pressing the Tab or Enter key, the button’s dimensions visibly change. You are going to set up the square button and three others like it as navigation buttons grouped in a square arrangement, so they must be relatively small.
17. Still in the Button Size inspector, change the struts to match those of the search field and the date picker. All of these are navigation buttons, so they should be grouped together in the lower-right corner of the window.
18. Hold down the Option key and drag the square button downward. In Interface Builder, Option-drag creates an identical copy of a user interface element, including its springs and struts settings, and places it wherever you drop it.

Do the same thing two more times, until you have four buttons arranged in a square with their edges touching. The buttons are identical, including the springs and struts settings.

19. Select all four square buttons. You can do this by clicking one of them and Shift-clicking the other three, but it is easier to drag a selection rectangle until all four are selected. Drag the group until a guide shows you that it is the correct distance to the left of the search field and the date picker. The height of the group of square buttons does not exactly match the height of the search field and the date picker combined. With the four square buttons still selected as a group, press the up and down arrow keys to nudge the group up and down until it appears centered.
20. Now comes the interesting part. These four square buttons are to include graphic elements indicating their functions. The user will understand them at a glance if you apply up and down arrows similar to those on a DVD or CD player, but where are you going to find images like that? This is a perennial problem for software developers, few of whom are endowed with sufficient artistic talent to draw effective graphics. See the “How to Obtain Graphic Images” sidebar for some suggestions.

How to Obtain Graphic Images

One of the things that make Mac OS X so attractive is all those gorgeous icons and other graphics on the desktop, in toolbars, in the Dock, and just about everywhere on the screen. Unfortunately for most of us, they aren’t easy to come by. Coding skill and artistic talent don’t seem to be delivered in the same package very often.

What do you do when you’ve finished writing a brilliant program and you’re ready to take it to market? Stick figures and geometric diagrams just don’t work as marketing tools.

The short answer is that you have to hire someone. This is also the realistic answer for any developer with ambitions to make it big in the marketplace.

For the rest of us, there are other approaches. One approach that has promise is to use Apple’s own images. Apple has begun to make system icons and images available for use by developers through the Media tab of Interface Builder’s Library window. Unfortunately, the Library window doesn’t include very many system images at this time. The Media tab seems to be focused more on making it easy for you to access your own images or those you have purchased. One can only hope that Apple will make everything on the system available through the Media tab.

(continues on next page)

How to Obtain Graphic Images *(continued)*

In addition, many Web sites offer free images for your use. Most of them, to my taste, are way too cute. But some of them are very good. It is definitely worth your time to search for a solution in this domain. There are also Web sites that offer images for sale.

For buttons, it is often the case that geometric figures work perfectly well. The navigation buttons in the Vermont Recipes application's diary window are an example. Almost everybody is familiar with the graphic images on DVD or CD player buttons. They consist of simple triangles and bars. Anybody with an appropriate drawing application can create them.

The requirements for button images are straightforward. Mac OS X is moving slowly into the age of resolution independence, and pixel-based images are inappropriate in that context. A small pixel image looks great on the screen at its normal resolution, but as soon as you enlarge it, the pixelation effects make it very ugly. For that reason, Apple's documentation insists that you should use scalable vector graphics for button images. One file format that works well in this context is scalable PDF images.

Unfortunately, many of the applications that make it easy to draw scalable vector images, such as Adobe Photoshop, are rather expensive. I'm no expert in this field, but I've searched the Web. I urge you to do the same. There appear to be a number of free or inexpensive applications that will serve very nicely.

For the navigation button images in this recipe, I happen to have used OmniGroup's OmniGraffle application, simply because I already have it. It has a very-easy-to-use vector graphics drawing interface, and you can save your creations as scalable PDF images. A large collection of images is available on Graffletopia (www.graffletopia.com). For this recipe, however, I drew my own. Somehow, a triangle and a horizontal bar didn't seem beyond my capabilities.

See Apple's *Resolution Independence Guidelines* for more information.

- a. For this book, I have created my own arrow images. In order to follow along with these instructions, download the Vermont Recipes project from the book's Web site and locate the four arrow images in the project folder. They are named *ArrowBottom.pdf*, *ArrowDown.pdf*, *ArrowTop.pdf*, and *ArrowUp.pdf*. Drag each of them into your working Vermont Recipes project folder, leaving them at the top level of the folder. If you don't have access to the book's Web site, find any PDF images and rename them to match these arrow names. They should be scalable vector PDF images.
- b. In Xcode, select the Resources group in the Groups & Files pane, and then choose Project > Add to Project. In the Open panel, navigate to the project

folder, select all four images, and click Add. Set up the second sheet as you have done several times before and click Add. The four arrow images appear in the Resources group.

- c. The Resources group is getting a bit full, so create a new group within it and place the four arrow images in the new subgroup. One way to do this is to select all four images and then use the contextual menu on them and choose Group. A new subgroup is created for you with its default name selected for editing. Enter **Images**. The Group command placed the four arrow images in the subgroup for you.
 - d. Go back to Interface Builder and select the top-left square button. In the Button Attributes inspector, open the Image pop-up menu, and there you see your four buttons. Choose ArrowTop. The upward-pointing arrow with a bar across the top appears in the button in the diary window. Because the image is a scalable vector PDF file, it is properly scaled with no effort on your part. To control scaling, use the Scaling pop-up menu. It is set by default to Proportionally Down, which works for these arrow buttons. If you used an image that is too small, change the setting to “Proportional Up or Down.” Go through the same exercise with the other three square buttons, placing the ArrowBottom image in the lower-left corner, the ArrowUp image in the upper-right corner, and the ArrowDown image in the lower-right corner.
21. To make sure the controls are positioned properly relative to one another and the sides and bottom of the window, move them around until you are satisfied that text baselines line up across the window, edges are aligned vertically, and margins comply with the guides to the extent that the other alignments allow. In the case of the navigation buttons, line up the images, not the borders; you will turn them into borderless buttons in a moment. Finally, adjust the placement of the bottom edge of the split view. Command-drag the bottom edge of the split view until the guides show that you have left the proper amount of space between it and the uppermost controls.
 22. Run the Cocoa Simulator now and resize the diary window. If you make it narrow enough, you notice that the controls overlap one another. To prevent this from happening, set the minimum width of the window to a value that leaves a reasonably wide space between the push buttons on the left and the new navigation buttons on the right. Apple’s *Human Interface Guidelines* counsel that white space is one of the most effective tools to inform the user of functional groupings in the user interface. The two push buttons on the left insert new material into the diary’s text, while the controls grouped on the right relate to navigation and selection of text.

The easiest way to set the window’s minimum width is to resize it in Interface Builder. Hold down the Command key to make sure you resize and reposition

all of the window's internal views and controls at once while you resize the window. When you have resized the window to the desired minimum size, turn to the Window Size inspector, select the Minimum Size checkbox, and click its Use Current button. The numbers in the Width and Height fields change to reflect the current size of the window. Use the Cocoa Simulator to confirm that it can no longer be resized to a smaller size.

Now that you've finished with the Cocoa Simulator, go back to the Button Attributes inspector and deselect the Bordered checkbox in the Visual section for all four navigation buttons. The window looks less cluttered without the square borders outlining the images, and the images are more easily understood if they stand free.

23. Finally, you should address the order in which the Tab key selects controls and other views. This is known as the window's *tab order*. Most windows have an *initial first responder*, and each of its views has a *next key view*. You use the `initialFirstResponder` and `nextKeyView` outlets to connect the views in the window in a complete circle known as the *key view loop*. If you don't set up the key view loop yourself, Cocoa does a reasonable job of guessing, but you shouldn't leave this to chance.

The typical user expects to begin typing in the diary window's text view after opening the diary window, without first having to click in the text view to select it for editing. Therefore, you should designate the text view in the top pane of the split view as the initial first responder.

From the initial first responder, tabbing proceeds from view to view in the window in an order that you can determine. Tabbing automatically skips views that are currently disabled. Within a complex control like the date picker, tabbing is already set up for you to move from element to element within the control in an appropriate order. To tab out of a text view, the user must press Control-Tab, since pressing Tab alone inserts a tab character into the text view. This is not true of text fields such as the search field, since tabs normally cannot be inserted in them. In System Preferences, users can elect to tab between views of any kind, not just text views and text fields, so you must always set the tab order for all of them.

The tab order of the views in the window should proceed roughly from top to bottom and left to right, but it is important to maintain functional groupings. A sensible tab order in the diary window is to start with the text view in the top pane of the split view, and then proceed to the Add Entry button, then to the Add Tag button, and then over to the group of navigation controls on the right. In that group, tabbing should select the top and bottom arrows, then the up and down arrows, and finally the date picker and the search field in that order. The last control should lead back to the text view, because tab order must always form a full circle.

The text views in the top and bottom panes of the split view are interesting. I suggest that the user should not be able to tab from the top text view to the bottom text view because the bottom one is usually collapsed. To a user pressing Control-Tab to tab out of the top text view, it would appear that nothing happened, and the Add Entry button would be selected only with a second press of Control-Tab. Both text views display the same text storage object, so even if both of them are expanded, a user does not need to tab from the top one to the bottom one. But if the user happens to be typing in the bottom pane, its next key view should be the Add Entry button, just as the top pane's next key view is the Add Entry button.

Start by selecting the diary window. In the Window Connections inspector, drag from the marker next to the `initialFirstResponder` outlet to the upper part of the top pane in the diary window's split view. You know you have selected the text view embedded in the scroll view when the term *Text View* appears in the window.

Next, select the text field in the top pane of the split view, and drag from the `nextKeyView` outlet to the Add Entry button. Do the same from the text field in the bottom pane of the split view, putting the nib file's window into outline or browser mode to make it easy to select the bottom text view without having to reposition the divider in the diary window. Select each remaining control in the window in turn and connect its `nextKeyView` outlet to the next control, ending back at the top text view. An alternate way to do this for each control is to Control-drag from one control to the next and select the `nextKeyView` outlet in its HUD.

24. Save the nib file, and then run the Cocoa Simulator to ensure that the new controls behave properly as you resize the window.

Step 2: Implement the Add Entry Push Button

You learned in Step 7 of Recipe 2 how to connect an existing action method to a button to make the button respond when the user clicks it. Creating a stub for a custom action method and hooking it up is easy. The hard part in this step and the next will be figuring out how to write the body of the `-addEntry:` and `-addTag:` action methods. Start with the Add Entry button.

1. Open the `DiaryWindowController.h` header file in Xcode. Add this action method declaration above the `@end` directive:
 - `(IBAction)addEntry:(id)sender;`

Open the `DiaryWindowController.m` source file. Insert the following stub of the `-addEntry:` action method definition between the existing `-otherDiaryView` and `-windowDidLoad` method definitions.

```
- (IBAction)addEntry:(id)sender {  
}
```

The signature of every action method follows the same pattern, as you saw when you wrote the `-newDiaryDocument:` action method in Step 5 of Recipe 3. The `IBAction` return type is a synonym for `void`, and it informs Interface Builder that this is an action method. Every action method takes a single parameter, the `sender`, usually typed as `id` for maximum flexibility.

When you write the body of an action method, you are free to ignore the `sender` parameter value, but it can be very useful. Developers often forget that the `sender` parameter is available in an action method. They go to extraordinary lengths to build up a reference to the object that sent the action message, when all along the `sender` was right there begging to be used. For example, you can determine whether the `sender` was a button or a menu item and, if it was a complex control, what the settings of its constituent cells were after its action method was sent.

2. Open the `DiaryWindow` nib file in Interface Builder. Control-drag from the Add Entry button in the diary window to the First Responder proxy in the `DiaryWindow` nib file's window, and then select the `addEntry:` action in the Received Actions section of the HUD. The Add Entry button is now connected. Clicking it while the application is running executes the `-addEntry:` action method.

Save the nib file once you've done this.

3. Several techniques can be used to verify that an action method is properly connected. One is to put a call to Cocoa's `NSLog()` function in the body of the action method, then build and run the application and watch the debugger console while you click the button. To do this now, add this statement to the `-addEntry:` stub implementation in the `DiaryWindowController.m` source file:
`NSLog(@"Add Entry button was clicked.");`

Choose `Run > Console` and position the Debugger Console window so that you can see it while the application is running. Then build and run the application, choose `File > New Chef's Diary`, and click the Add Entry button. If the action method is properly connected, you see a message in the Debugger Console showing the date and time, the name of the application with some cryptic information about it, and the message you specified in the `NSLog()` function call.

The `NSLog()` function call would be more useful if it identified what you clicked by using the `sender` parameter value instead of hard coding it in the string. You will add an Add Entry menu item to the menu bar in Recipe 5, and it would be

disconcerting to see a console message that the Add Entry *button* was clicked when you had actually chosen the Add Entry *menu item*. Change the `NSLog()` function call to this:

```
NSLog(@"Add Entry; sender is %@ named \"%@\"", sender, [sender title]);
```

Now the debugger console shows the title of the button as well as its class. This will work with a menu item, too, because menu items also respond to the `-title` message.

The string in the first parameter of the `NSLog()` function is called a *format string* because it can optionally contain placeholders. At run time, the values of the following parameters replace the placeholders in order. The two placeholders (`%@`) in the format string here are filled in at run time with the sender's object and its title. The `%@` placeholder is for object values only, including `NSString` object values. Different placeholders are available for a variety of other data types. Bookmark the "String Format Specifiers" section of Apple's *String Programming Guide for Cocoa*. It lists all of the available placeholders, and you will consult it often. If you use the wrong placeholder, the resulting type mismatch may generate its own runtime error, and, ironically, your debugging effort may itself be difficult to debug.

4. Now you're ready to write the body of the `-addEntry:` action method. Consider first what it should accomplish. It may be called in two situations: where the text of the diary window is empty, and where there is already some text in the window. The Add Entry button should append the title of a new diary entry to the end of whatever text is currently in the diary, starting on a new line if necessary, or simply insert it if the diary is empty. The title should be the current date in human-readable form, preceded by a special character reserved to mark the beginning of a diary entry. You will use the special marker character later to search for the beginning of diary entries, so it must be a character that the user would never type. The title should be in boldface so that it stands out visually. Undo and redo must be supported. Finally, the action method should end by inserting a new line so that the user can immediately begin typing the new entry.

In writing the `-addEntry:` action method, consider the MVC design pattern. Details regarding the structure and content of the diary, such as the fact that the diary is organized into entries having titles in the form of human-readable date and time strings and the special character used to mark a diary entry, belong in the MVC model. You should place methods relating to these features of the Chef's Diary in the `DiaryDocument` class. In the case of an entry's title, you therefore implement methods in `DiaryDocument` to return the title itself and the special marker character that precedes it, as well as a method that constructs the white space around the title that is required for proper spacing above and below it. These methods are `-entryMarker`, `-entryTitleForDate:`,

and `-entryTitleInsertForDate:`. In the course of writing the action method, you call `-entryTitleInsertForDate:` to obtain the fully constructed insert to be added to the diary.

Before you begin, delete the `NSLog()` function call you wrote earlier because you no longer need it.

Now implement the `-addEntry:` action method in the `DiaryWindowController.m` implementation file, as set out here:

```
- (IBAction)addEntry:(id)sender {
    NSTextView *keyView = [self keyDiaryView];
    NSTextStorage *storage = [keyView textStorage];
    NSString *titleString = [[self document]
        entryTitleInsertForDate:[NSDate date]];

    NSRange endRange = NSMakeRange([storage length], 0);
    if ([keyView shouldChangeTextInRange:endRange
        replacementString:titleString]) {
        [storage replaceCharactersInRange:endRange
            withString:titleString];
        endRange.length = [titleString length] - 1;
        [storage applyFontTraits:NSBoldFontMask range:endRange];
        [keyView didChangeText];
        [[keyView undoManager] setActionName:
            NSLocalizedString(@"Add Entry",
                @"name of undo action for Add Entry")];

        [[self window] makeFirstResponder:keyView];
        [keyView scrollRangeToVisible:endRange];
        [keyView setSelectedRange:
            NSMakeRange(endRange.location + endRange.length + 1, 0)];
    }
}
```

The first block sets up three local variables, `keyView` to hold the text view in whichever pane of the diary window currently has keyboard focus; `storage` to hold the view's text storage object; and `titleString` to hold the entry title insert obtained from the diary document. Foundation's `+[NSDate date]` method returns an `NSDate` object representing the current date and time.

You define `keyView` to specify which of the two text views in the split pane currently has keyboard focus. You do this by calling a utility method you will write shortly, `-keyDiaryView`, which uses `NSWindow`'s `-firstResponder` method. It is important to use the correct view, because at the end of the `-addEntry:` method

you scroll the new entry title into view and select it. These visual changes should occur in the pane the user is editing. You use `keyView` throughout the `-addEntry:` method as a convenient way to refer to the text view.

The documentation for Cocoa's text system recommends that you always perform programmatic operations on the contents of a text view by altering the underlying text storage object. Although `NSTextView` implements a number of methods that alter the contents of the view without requiring you to deal directly with the text storage, such as `-insertText:`, you can't use them here because they are designed to be used only while the user is typing.

In `-addEntry:`, you get the diary's text storage object through its text view object, which is described in the *NSTextView Class Reference* as the front end to the Cocoa text system. Apple's *Text System Overview* goes so far as to say that most developers can do everything using only the `NSTextView` API. You could just as easily have called the `-diaryDocTextStorage` accessor method in `DiaryDocument` to get the text storage object, but MVC principles make clear that it is preferable to use methods that don't require the window controller to know anything about specific details of the document's implementation.

You set the `titleString` local variable by calling another method you will write shortly, `-entryTitleInsertForDate:`.

The next block uses `NSMutableAttributedString`'s `-replaceCharactersInRange:withString:` method, inherited by `NSTextStorage`, to insert the entry title at the end of the Chef's Diary.

The call to `-shouldChangeTextInRange:replacementString:` that precedes this operation tests whether changing the contents of the text storage is currently permitted. You should always run this test before changing text that is displayed in a text view, even if you're confident that changing the text is appropriate. You—or Apple—might later decide to override `-shouldChangeTextInRange:replacementString:` to return `NO` under circumstances that you haven't anticipated. By default, it returns `NO` only when the text view is specified as noneditable.

These two methods, like many methods in the Cocoa text system, require an `NSRange` parameter value identifying the range of characters to be changed. You're going to need the range in other statements, as well, for example, to scroll the text view to the newly inserted title. You define it just before the test as a range of zero length located at the end of the existing text, if any, storing it in the `endRange` local variable.

The method then revises the length of `endRange` to reflect the fact that you have now added the title string to the end of the document. It uses this new range to apply the `NSBoldFontMask` font trait to the title. You exclude the trailing newline character from the boldface trait to ensure that the text the user begins typing after adding the title is not boldface.

The method next uses a standard text system technique to make the user's insertion of a new diary entry undoable. Until now, you have operated under the assumption that text views automatically implement undo and redo because you selected the Undo checkbox in the Text View Attributes inspector in Interface Builder. Your assumption is correct, but only as long as the user performs built-in text manipulation operations. When you implement a custom text manipulation operation that isn't part of `NSTextView`'s built-in typing repertoire, you have to pay attention to undo and redo yourself.

Whenever you implement a method, such as the `-addEntry:` method here, that gives the user the ability to perform a custom operation in a text view, you must bracket the code with calls to `-shouldChangeTextInRange:replacementString:` and `-didChangeText`. This ensures that all the expected notifications get sent and all the expected delegate methods get called, and it ensures that the operation is undoable and redoable.

The titles of the Undo and Redo menu items should reflect the nature of the undo or redo operation. You accomplish this by passing the string `@ "Add Entry"` to the undo manager's `-setActionName:` method, using the `NSLocalizedString()` macro to make sure your localization contractor can translate the string for use in other locales. In the English locale, the menu item titles will be Undo Add Entry and Redo Add Entry.

In the final block, you scroll the title into view. In case the user has set the Full Keyboard Access setting to "All controls" in the Keyboard Shortcuts tab of the Keyboard pane of System Preferences, you also set the current keyboard focus on the text view in the top pane of the split view, unless it was already on the bottom pane. This respects the user's expectation that clicking the Add Entry button enables typing immediately below the new entry's title, even if some other view in the window, such as the search field, had keyboard focus.

5. Write the `-keyDiaryView` method called at the beginning of `-addEntry:`. This is a very short method and its code could just as well be written in line. However, you will need it in several places so it's a good candidate for a separate method.

At the end of the `DiaryWindowController.h` header file, insert this declaration:

```
- (NSTextView *)keyDiaryView;
```

In the `DiaryWindowController.m` implementation file, define it like this:

```
- (NSTextView *)keyDiaryView {
    return ([[self window] firstResponder] == [self otherDiaryView]) ?
        [self otherDiaryView] : [self diaryView];
}
```


If the user is currently typing in the bottom pane of the diary window, this method returns `otherDiaryView`. If any other view in the window has keyboard focus, it returns the primary text view, `diaryView`, which is in the top pane of the window.

6. Next, add the `-entryTitleInsertForDate:` method and its two supporting methods to the `DiaryDocument` class. They belong in `DiaryDocument`, not `DiaryWindowController`, because they provide information about the structure and content of the document.

In the `DiaryDocument.h` header file, declare them as follows:

```
- (NSString *)entryMarker;  
- (NSString *)entryTitleForDate:(NSDate *)date;  
- (NSString *)entryTitleInsertForDate:(NSDate *)date;
```

In the `DiaryDocument.m` implementation file, define them as follows:

```
- (NSString *)entryMarker {  
    return DIARY_TITLE_MARKER;  
}  
- (NSString *)entryTitleForDate:(NSDate *)date {  
    NSString *dateString;  
    if (floor(NSFoundationVersionNumber)  
        <= NSFoundationVersionNumber10_5) {  
        NSDateFormatter *formatter = [[NSDateFormatter alloc] init];  
        [formatter setDateStyle:NSDateFormatterLongStyle];  
        [formatter setTimeStyle:NSDateFormatterLongStyle];  
        dateString = [formatter stringFromDate:date];  
        [formatter release];  
    } else {  
        dateString = [NSDateFormatter localizedStringFromDate:date  
                      dateStyle:NSDateFormatterLongStyle  
                      timeStyle:NSDateFormatterLongStyle];  
    }  
    return dateString;  
}  
  
- (NSString *)entryTitleInsertForDate:(NSDate *)date {  
    NSTextStorage *storage = [self diaryDocTextStorage];  
    NSString *leadingWhitespaceString;  
    if ([storage length] == 0) {  
        leadingWhitespaceString = @"";
```

(code continues on next page)

```

    } else if ([[storage string] hasSuffix:@"\n"]) {
        leadingWhitespaceString = @"\n";
    } else {
        leadingWhitespaceString = @"\n\n";
    }
    NSString *formatString = @"%@@@ %@\n";
    return [NSString stringWithFormat:formatString,
        leadingWhitespaceString, [self entryMarker],
        [self entryTitleForDate:date]];
}

```

The first method, `-entryMarker`, simply returns `DIARY_TITLE_MARKER`. Don't forget to define it. At the top of the `DiaryWindowController.m` source file, add this line below the `#import` directives:

```
#define DIARY_TITLE_MARKER @"\u270E" // Unicode LOWER RIGHT PENCIL
```

You can now use the `DIARY_TITLE_MARKER` macro anywhere in this file's code, and the preprocessor automatically substitutes the string. You return it in a method as well, in case you need it in the diary window controller.

The second method, `-entryTitleForDate:`, formats the current date and time into a string suitable for use as the title of a new diary entry. You should recognize the opening `if` test from Step 3 of Recipe 2, where you learned that the statements in the first branch are executed only when the application is running under Leopard and not under Snow Leopard. In fact, these statements would work correctly under Snow Leopard, too. However, a new convenience method was introduced in Snow Leopard that does the same thing in a single statement. This book is focused on Snow Leopard, so you should know how to use it. Because the new method is in Foundation, not AppKit, you test against `NSFoundationVersionNumber10_5`.

Although there are other ways to create a date string, the Cocoa documentation recommends that you use `NSDateFormatter`, a built-in Cocoa class. Here, you allocate and initialize a date formatter object using its designated initializer, `-init`, and then you set both its date style and its time style to `NSDateFormatterLongStyle`, one of several date and time styles declared in Cocoa. Then you tell the formatter to create and return a string reflecting these styles by calling the formatter's `-stringFromDate:` method. When that is done, you release the formatter because you're through with it.

When Vermont Recipes is running under Snow Leopard, all this work is done by a single new method, `+localizedStringFromDate:dateStyle:timeStyle:`. It is a class method that you can call globally without allocating and initializing your own date formatter. You don't release the formatter afterward because you didn't create it.

Both techniques set up the date string according to the date and time styles you specified. The styles control which date and time elements appear in the string and whether short or long forms are used. The user's locale and current date and time formatting preferences are honored, as set in the Language & Text pane in System Preferences.

The third method, `-entryTitleInsertForDate:`, sets up the `leadingWhitespaceString` local variable. It then assembles it, the marker character, and the entry title into a combined string suitable to be inserted into the Chef's Diary. The `leadingWhitespaceString` string is used to ensure that there is always a blank line before a new diary entry's title. If the text field is currently empty, of course, there should be no white space before the initial entry's title. If it is not empty, the application specification calls for a new title to be added at the end of the text view's content. If the last character in the text view is a newline character, one more newline character is added. Otherwise, two newline characters are added. This ensures that a blank line always separates a new entry's title from a preceding entry's text.

The method then uses NSString's `+stringWithFormat:` class method. This is a workhorse that you will use very frequently to compose formatted strings. It takes a variable number of parameters, the first of which is a *format string* like the format string you used in the `NSLog()` function earlier. The remaining parameters generate objects or other types of data and place them into the corresponding placeholders embedded in the text of the format string. Here, you define a format string with three placeholders and some whitespace. After assigning the format string to a local variable, `formatString`, you pass it and other information into the `+stringWithFormat:` method.

Examine the format string. The first placeholder is `%@`. You saw it in the `NSLog()` function a moment ago. Here, you place the `leadingWhitespaceString` object in the placeholder, even if it is an empty string.

The second placeholder is also `%@`, separated from the first placeholder by a space character that will appear in the final string. You replace the second placeholder with the entry title marker character, which you defined earlier as an NSString object holding the Unicode LOWER RIGHT PENCIL character, code point 0x270E. This is the special character you will use to mark the beginning of each diary entry in the diary window.

Pause for a moment to consider this second placeholder. If you were building the Vermont Recipes application under Leopard, you would probably write the format string as `@ "%@%C %@\n"`, and you would define `DIARY_TITLE_MARKER` not as a string but as the hexadecimal number `0x270E`. The second placeholder would be `%C`, not `%@`. Look it up in the "String Format Specifiers" section of Apple's *String Programming Guide for Cocoa*, which you bookmarked a moment

ago, and you see that the `%C` placeholder is designed to hold a 16-bit Unicode code point.

When you build an application under Snow Leopard, you don't have to use `%C` and the numeric code point for a Unicode character. Xcode 3.2 in Snow Leopard compiles code using the C99 dialect of the C programming language by default, and C99 lets you define strings using an escape sequence for Unicode characters like this: `@"\u270E"`. The compiled C99 object code runs perfectly well under Leopard and Snow Leopard. In fact, you could even build the application under Leopard using the escape sequence, if you set up the project to use the C99 dialect.

An easy way to find Unicode characters is Apple's Character Viewer. Add it to your menu bar by opening the Keyboard tab of the Keyboard pane in System Preferences and selecting the "Show Keyboard & Character Viewer in menu bar" setting. In many applications, you can open it by choosing Edit > Special Characters. Choose it from an application or choose Show Character Viewer from the menu bar item, choose Code Tables from the View pull-down menu, and scroll to Unicode block 00002700 (Dingbats).

The third placeholder in the format string is another `%@` placeholder. Here, you replace it with the title string returned by the `-entryTitleForDate:` method you just wrote.

The format string contains the escape sequence `\n` at the end. This escape sequence also appears in the `whitespaceString` variable. It inserts the standard newline control character into the string. You could just as well have used `\r` for the carriage return control character, or `\r\n` for the standard Windows line-ending control character sequence. When you use this string to create an attributed string for insertion into the text view, Cocoa automatically interprets any of these escape sequences as a signal to begin a new paragraph in the text view. You should make a habit of using one of them consistently, however, because you must sometimes search for it.

7. If you build and run the application now, you find that the Add Entry button works exactly as expected. Try it. Open a new Chef's Diary window and click Add Entry. A new boldface title appears in the text view. A cute little pencil image appears at the beginning of the title. Type some more text, and then click Add Entry again. Another title appears at the end, separated by a blank line from the preceding text. Choose Undo Add Entry and Redo Add Entry, and see that undo and redo work.

Try a different sequence of actions. Open a new Chef's Diary window, and then click Add Entry. Then save the document and click Add Entry again. Choose Undo Add Entry, and the second entry title goes away, leaving the document

marked clean because it is now back in the state it was in when you saved it. Open the Edit menu again, and you see an Undo Add Entry menu item. Choose it, and the first title is removed, leaving the document marked dirty because it now differs from the document as you saved it. Choose Redo Add Entry, and the title is reinserted, leaving the document marked clean because you are now back at the point where you saved it. You have just undone an action back in time past the save barrier. This ability was introduced in Mac OS X 10.4 Tiger and is now standard. Some applications, such as Xcode, present an alert asking whether you really want to undo past the save barrier, but most applications don't do this because users have come to expect it. The ability to undo past the save barrier does not survive quitting the application.

8. In the course of playing with the text view, you might discover one inconsistency regarding undo past the save barrier. Try typing some text in the Chef's Diary window, then save the document, and then type some more text. Now choose Edit > Undo Typing. All of the text you typed, both before and after the point where you saved the document, is removed at once, and choosing Edit > Redo Typing restores all of it.

The typical user expects that saving a document will interrupt the collection of undo information. After you save a document and type some more text, Undo Typing should undo only back to the point at which the document was saved. To undo the typing before the save operation should require another invocation of Undo Typing.

According to Apple's *NSTextView Class Reference*, you should resolve this issue by invoking `-breakUndoCoalescing` every time the user saves the document.

Apple does this in its TextEdit sample code by overriding `NSDocument`'s implementation of the `-saveToURL:ofType:forSaveOperation:error:` method. The problem with TextEdit's implementation, as the TextEdit sample code notes in a comment, is that undo coalescing is broken not only when the user saves a document, but also when the document saves itself because autosaving is turned on. The TextEdit sample code points out that this is potentially confusing to the user, who normally won't notice the autosave operation.

To avoid this problem, override `-saveToURL:ofType:forSaveOperation: delegate: didSaveSelector:contextInfo:`, instead. You could override the same method that TextEdit overrides and test its `saveOperation` argument to exclude `NSAutoSaveOperation` operations, as you do here, but this is an opportunity to take advantage of the temporary delegate callback design pattern common in Cocoa. This method is called for all kinds of save operations, including those performed by the Save and the Save As menu items as well as autosave operations. You test for an autosave operation—which you will turn on in Recipe 7—and exclude it from the callback method so that autosaves do not break undo coalescing. To

understand the message flow when a document in a document-based application is created, opened, or saved, see the figures in the “Message Flow in the Document Architecture” section of Apple’s *Document-Based Applications Overview*. They contain lists of methods you can override. See also the “Autosaving in the Document Architecture” section.

Another problem with the `TextEdit` sample code is that it accesses the document’s array of window controllers to call an intermediate `-breakUndoCoalescing` method written specifically to make this technique work. The intermediate method in turn calls `NSTextView`’s `-breakUndoCoalescing` method. It is generally preferable to avoid requiring the document to know anything about how the window controller is implemented.

To avoid this problem in the diary document, take advantage of the Cocoa text system by using the document’s `NSTextStorage` object to locate all of its associated `NSTextView` objects and call their `-breakUndoCoalescing` methods.

In the `DiaryDocument.m` source file, implement these two methods:

```
- (void)saveToURL:(NSURL *)absoluteURL ofType:(NSString *)typeName
    forSaveOperation:(NSSaveOperationType)saveOperation
    delegate:(id)delegate didSaveSelector:(SEL)didSaveSelector
    contextInfo:(void *)contextInfo {
    NSNumber *saveOperationNumber =
        [NSNumber numberWithInt:saveOperation];
    [super saveToURL:absoluteURL ofType:typeName
        forSaveOperation:saveOperation delegate:self
        didSaveSelector:@selector(document:didSave:contextInfo:)
        contextInfo:[saveOperationNumber retain]];
}

- (void)document:(NSDocument *)doc didSave:(BOOL)didSave
    contextInfo:(void *)contextInfo {
    NSSaveOperationType saveOperation =
        [(NSNumber *)contextInfo intValue];
    [(NSNumber *)contextInfo release];
    if (didSave && (saveOperation != NSAutosaveOperation)) {
        for (NSLayoutManager *manager
            in [[self diaryDocTextStorage] layoutManagers]) {
            for (NSTextContainer *container in [manager textContainers]) {
                [[container textView] breakUndoCoalescing];
            }
        }
    }
}
```


This is your first contact with a common Cocoa design pattern, the callback selector to a temporary delegate. The `delegate` argument of the first method allows you to designate any object as the method's temporary delegate. Here, as is commonly the case, you designate `self` as the temporary delegate. This relationship is only for purposes of the callback method, and it lasts only until the callback method is called.

The `didSaveSelector:` parameter allows you to identify a selector for the callback method, which you must implement in the temporary delegate, `self`. Cocoa will call the callback method identified by the selector automatically when some event occurs. The documentation describes the event that triggers the callback method, and it also specifies the required signature of the callback method. The `-saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo:` method is documented to call the callback method when the save operation completes successfully.

Here, the `-saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo:` method does only one thing: It calls super's implementation of the method, designating `self` as the temporary delegate and `document:didSave:contextInfo:` as the callback selector. The call to super starts the save operation. It also specifies the callback selector, using the `@selector()` compiler directive. In addition, it encapsulates the `saveOperation` argument, an integer of type `NSSaveOperationType`, in an `NSNumber` object and passes it to the callback method in the `contextInfo` parameter. You retain the `NSNumber` object before passing it to the callback method, then release it in the callback method when you're finished with it.

The callback method is where the important action takes place. First, it tests the `didSave` argument, since you don't want to break undo coalescing if the save operation is not successful. Next, it extracts that `saveOperation` value from the `contextInfo` argument and checks whether it is an autosave operation. If the save operation did complete and it was not an autosave operation, the method uses nested `for` loops to traverse all of the `NSLayoutManager` objects, all of the `NSTextContainer` objects, and finally all of the `NSTextView` objects in the diary document's text storage object. It calls `-breakUndoCoalescing` on each of the text views it finds. This logic covers all possible text system arrangements, so it makes no assumptions about the window controller or the user interface of the application.

Try the same experiment now. Type some text, save the document, type some more text, and choose `Edit > Undo Typing`. Only the text added after the document was saved is removed. Choose `Undo Typing` again, and the rest of it is removed. Choose `Redo Typing` twice, and the text comes back in two discrete steps. The document is marked dirty or clean correctly based on the point at which it was last saved. This works correctly whether you save the document using the `Save` or the `Save As` menu item.

Step 3: Implement the Add Tag Push Button

The Add Tag button works like the Add Entry button, with two exceptions. One exception is that the special character marking an entry's tag list is different from the character marking entry titles, to allow separate searches for titles and for tags. The other is that the tag list is inserted immediately following the title of whichever diary entry currently has keyboard focus in the split view; that is, the diary entry in which the text insertion cursor is currently located.

The tag list is actually a paragraph of text starting with the special character for tags followed by a space, the word *Tags*, a colon, another space, and tag words or phrases typed by the user. The Chef's Diary is not a database with records and fields. It is a simple RTF text object, and the tags can be edited like any other text. Special characters are used to identify the entry titles and the tags simply to make it easy to locate them programmatically. Entering the actual tags in the tag list is up to the user, who simply types them.

In Vermont Recipes, individual tags can be separated by any white space or punctuation marks, such as commas and spaces. This is done mostly for the sake of simplifying the code, but it works rather well. For example, if the tag list is *ho hum*, the user can use the search field to search for *ho*, *hum*, or *ho hum*. The problem with this is that the first two characters of *hotel* in a tag list will also be found. In a real-world application, you should use a more sophisticated approach. For now, just make a note of it.

A significant difference in the code for the new `-addTag:` action method is dictated by the need to find the title of the current diary entry so that the tag list can be inserted immediately following it, instead of at the end of the text view. A newline character is always inserted before the tag list because the tag list is always inserted at the end of the current title, before the title's trailing newline character. The tag list has no trailing newline character because the user should be allowed to start typing a new tag immediately following the *Tags:* label. If the entry already has a tag list, the method places the insertion pointer at its end, after the last tag, so that the user can immediately start typing additional tags.

Another difference in the `-addTag:` method is that it should look for the next diary entry, in order to define the end of the search range for an existing tag list in the current entry. This is necessary in part because the diary is a simple text file, not a database file, and the user might type additional text between a title and a tag list. Although the Add Tag button always inserts a new tag immediately following the title of the current entry, it should be user friendly and accommodate any user who has typed text between the current entry's title and an existing tag list. Another reason the search must stop at the next entry title marker is that tags are optional and the current entry may not have a tag list at all.

Looking ahead, you can see that the code to locate the title of the current and next entries might do double duty in the implementation of the navigation buttons, coming up shortly. It would therefore be a good idea to place this navigation code in separate methods. Call them `-currentEntryTitleRangeForIndex:` and `-nextEntryTitleRangeForIndex:`, and have them return both the location and the length of the title as an `NSRange` structure. Once you have written those two methods, you will be ready to write the method that returns the location and length of the current entry's tag list, if there is one, or the location where the tag list should be inserted and a length of 0 if there is no existing tag list in the current entry. You will call that method `-currentEntryTagRangeForIndex:`. Only after you have written these three supporting methods will you be ready to write the `-addTag:` action method.

As you work through the rest of this recipe, you may be struck by the fact that so many methods return an `NSRange` struct. The Cocoa text system relies heavily on ranges for efficiency and convenience. A text storage object can be very long, and you don't want to have to search the whole thing every time you need to find one substring or character in it.

Before getting started, consider again the MVC design pattern. The methods you are about to write start with information about the insertion point or the current selection in one of the diary window's text views, and they use that information to guide a search for characters in the diary document's text storage. They then add text to the text storage, and they end by changing the text view's selection and scrolling the new selection into view. These operations involve both the MVC model and the MVC view. You therefore implement some of them in the `DiaryWindowController` class and some of them in the `DiaryDocument` class. Searching for characters in the text storage object and adding tags to it is work for the MVC model, so you place these supporting methods in the `DiaryDocument` class, which, as you know, acts as a model controller.

Another thing to consider is code reuse. The methods you are about to write, and others like them that you will write later, involve some repetitive operations. You should look for every opportunity to gather repetitive code into separate methods for easy reuse.

Start with the `DiaryDocument` methods. I will reproduce one or two of these methods here. The rest are similar. Rather than reproduce all of them in the book, I refer you to the downloadable project file for Recipe 4 at www.peachpit.com/cocoarecipes, where they are laid out in full.

1. Just as you wrote the `-entryMarker`, `-entryTitleForDate:`, and `-entryTitleInsertForDate:` methods in Step 2 for the entry title, you need to write `-tagMarker`, `-tagTitle`, and `-tagTitleInsert` methods for the tag title. They are much simpler, because they don't have to construct any date or time strings. You will find them in the downloadable project file for Recipe 4.

2. Now you're ready to write the first of several methods that obtain the range of entry titles and tag titles in the Chef's Diary. You start with the current entry, which is defined as the entry in which the insertion point is currently located. In this method and all of the other methods like it that you have yet to write, the current insertion point is passed into the method as its index in the text storage object.

In the `DiaryDocument.h` header file, declare the `-currentEntryTitleRangeForIndex:` method at the end of the file, just before the `@end` directive:

```
- (NSRange)currentEntryTitleRangeForIndex:(NSUInteger)index;
```

In the `DiaryDocument.m` implementation file, define it like this:

```
- (NSRange)currentEntryTitleRangeForIndex:(NSUInteger)index {
    if ([[self diaryDocTextStorage] length] == 0)
        return NSRange(NSNotFound, 0);
    index = [self adjustedIndexIfMarkerAtIndex:index];
    NSUInteger markerIndex = [[[self diaryDocTextStorage] string]
        rangeOfString:[self entryMarker]
        options:NSBackwardsSearch
        range:NSMakeRange(0, index)].location;
    return [self rangeOfLineFromMarkerIndex:markerIndex];
}
```

The first statement takes care of an edge case. If the text storage object is currently empty, it doesn't contain an entry title, so you create and return a range with length 0 whose location is `NSNotFound`. This conforms to the Cocoa text system convention that a search that comes up empty returns a range whose `location` member is `NSNotFound`. This also simplifies the rest of the method's code by eliminating any need to test for an empty text storage object.

The second statement deals with another edge case. From the user's perspective, if the insertion point is currently located at the entry marker character—that is, immediately before the entry marker and on the same line—it is in the entry marked by that character. In other words, this is the current entry. You force this convention on the text system by defining the search range so that its `location` member is just after the entry marker. As a result, the backward search finds it immediately. This will also work for forward searches because it prevents the method from finding the marker at the insertion point and allows it to find the following marker for the next entry. You do this in a utility method, `-adjustedIndexIfMarkerAtIndex:`, which you will write shortly.

Finally, the method pursues an efficient strategy to locate the current entry's entry marker. You will soon follow a similar strategy in methods designed to search for the next and previous titles. In summary, you start by searching

backward from the insertion point (the `index` argument), looking for the first entry marker you find. If you don't find one, you know the current selection was in an untitled preface near the beginning of the diary, and you exit signaling that a title was not found. If you do find an entry marker, you next search forward from that point, looking for the first newline character you find, or for the end of the file if you don't find a newline character. The newline character or the end of the file defines the length of the entry's title. When returning a range value from the `-currentEntryTitleRangeForIndex:` method, the `location` member represents the location of the title marker, and the `length` member counts the marker and all the characters in the title up to, but not including, the newline character. You must exclude the newline character so that you can use the range even if it ends at the end of the file instead of at a newline character.

You carry out this strategy in two steps. First, you call an important Cocoa method, `-rangeOfString:options:range:`, to find the preceding entry marker. Then you call a utility method that you will write shortly, `-rangeOfLineFromMarkerIndex:` to find the first newline character following the entry marker, or the end of the text storage object's contents if there is no following newline character. The utility method converts this to a range encompassing the entry's title, and you return that range from the `-currentEntryTitleRangeForIndex:` method. This utility method deals specially with the case where no preceding entry marker is found, returning, as you by now expect, a range whose location member is `NSNotFound`. The application interprets this to mean that there is no current entry; instead, the insertion point is considered to lie in a preface to the diary preceding its first entry.

You perform the search for the preceding entry marker using `NSString`'s `-rangeOfString:options:range:` method. This workhorse text system method is highly optimized and very fast. What you search for is `DIARY_TITLE_MARKER`, using the `-entryMarker` method you just wrote. You defined the macro in Step 2 as a string containing the Unicode character with code point `0x270E`, a dingbat character representing a pencil.

If an entry marker is found, the `-currentEntryTitleRangeForIndex:` method next searches forward from the entry marker, looking for the newline character that defines the end of the entry's title. It does this using the `-rangeOfLineFromMarkerIndex:` utility method, which you will write in a moment.

3. Now write the two utility methods used by `-currentEntryTitleRangeForIndex:`. The first is `-adjustedIndexIfMarkerAtIndex:`.

Enter this declaration of the method just before the `-currentEntryTitleRangeForIndex:` method in the `DiaryDocument.h` header file:

```
- (NSUInteger)adjustedIndexIfMarkerAtIndex:(NSUInteger)index;
```

Enter the method's definition in the `DiaryDocument.m` implementation file:

```
- (NSUInteger)adjustedIndexIfMarkerAtIndex:(NSUInteger)index {
    return ((index < [[self diaryDocTextStorage] length]) &&
        ([[self diaryDocTextStorage] string] characterAtIndex:index]
        == [[self entryMarker] characterAtIndex:0]))
        ? index + 1 : index;
}
```

The method first tests whether `index` is less than the text storage's total length. This test returns **NO** if the insertion point is at the end of the diary, thus preventing the method from incrementing `index` to an illegal value. There is no point in searching for a marker character at the end of the file anyway, because it could not mark an entry title.

The method then tests whether the character at `index` is an entry marker, and, if it is, increments `index`. This implements the application's convention, described earlier, that if the insertion point is at the entry marker and on the same line, it is considered to be in the current entry.

4. Now write the second utility method just after the first.

In the `DiaryDocument.h` header file, declare the method as follows:

```
- (NSRange)rangeOfLineFromMarkerIndex:(NSUInteger)markerIndex;
```

In the `DiaryDocument.m` implementation file, define it like this:

```
- (NSRange)rangeOfLineFromMarkerIndex:(NSUInteger)markerIndex {
    if (markerIndex == NSNotFound) return NSRange(NSNotFound, 0);
    NSRange searchRange = NSRange(markerIndex,
        [[self diaryDocTextStorage] length] - markerIndex);
    NSUInteger newlineIndex = [[[self diaryDocTextStorage] string]
        rangeOfString:@"\n" options:0
        range:searchRange].location;
    if (newlineIndex == NSNotFound) return searchRange;
    return NSRange(markerIndex, newlineIndex - markerIndex);
}
```

The method returns a range with a location of `NSNotFound` and a length of 0 if the incoming `index` argument is `NSNotFound`. This is the case if the search in `-currentEntryTitleRangeForIndex:` found no preceding entry marker.

If there is a preceding entry marker, the method searches forward for a trailing newline character marking the end of the entry's title. Cocoa does not declare a constant for forward searches on the model of `NSBackwardsSearch` for backward searches. Just use 0.

The method has to take into account the possibility that the title is the last line in the diary; that is, that there is no following newline character. It therefore defines the search range to encompass the text storage from the entry marker to the end of the file. If a newline character is not found, it returns the search range, since without a trailing newline character the title must encompass the entire remainder of the text storage. Finally, if a newline character is found, the method returns the current title's range, including its location and its length excluding any trailing newline character.

5. You can now write the `-nextEntryTitleRangeForIndex:` method, which, like `-currentEntryTitleRangeForIndex:`, is needed before you can implement the `-currentEntryTagRangeForIndex:` and `-enterTag:` methods. You will find it in the downloadable project file for Recipe 4.

The `-nextEntryTitleRangeForIndex:` method is identical to `-currentEntryTitleRangeForIndex:` except that it searches the portion of the file following the insertion point and the search is forward toward the end of the file. There is no `NSForwardsSearch` constant like the `NSBackwardsSearch` constant, so you use `0`.

6. Now write the `-currentEntryTagRangeForIndex:` method. It is a little more complicated, because it has to search for the beginning and the end of the current entry before it can determine whether the current entry already has a tag title. To do this, it calls the `-currentEntryTitleRangeForIndex:` and `-nextEntryTitleRangeForIndex:` methods you just wrote. You will find it in the downloadable project file for Recipe 4.

The `-currentEntryTagRangeForIndex:` method is similar to `-currentEntryTitleRangeForIndex:` and `-nextEntryTitleRangeForIndex:`, but the strategy is a little different because the tag title is positioned differently in the text view. The application specification calls for an optional tag title to be inserted on the line immediately following an entry's title. You therefore start by searching backward from the insertion point for an entry marker. There should be no tag title in an untitled preface, and the returned range's `location` should therefore be `NSNotFound` if no entry marker is found searching backward. You perform the backward entry marker search using the `-currentEntryTitleRangeForIndex:` method you just wrote. If an entry marker is found, you next search forward for a following entry marker to define the endpoint of the tag marker search range. You use the `-nextEntryTitleRangeForIndex:` method you just wrote to perform the forward entry marker search. You then search the range between the two entry markers—or between the first entry marker and the end of the file—for a tag marker and its trailing newline character using code that is nearly identical to the searches for an entry marker and its trailing newline character in the methods you just wrote.

Another difference in the `-currentEntryTagRangeForIndex:` method is that it must return the location where a tag list should be inserted if there is no existing tag list in the current entry. In that case, you return a range with a `location` at the end of the current title string and a `length` of 0. In the `-addTag:` action method you are about to write, you will test for a `length` of 0 in order to choose between inserting a new tag list or simply moving the insertion point to the end of an existing tag list so the user can begin typing tags.

7. The `-addTag:` action method calls one more utility method that you have yet to write, `-insertionPointIndex:`. You have read several times that methods you have just written, like `-currentEntryTitleForindex:`, take as their `index` argument the current insertion point in the text view having keyboard focus. Write the method that gets the insertion point now. It belongs in the `DiaryWindowController` class because it relates to the MVC view.

At the end of the `DiaryWindowController.h` header file, insert this declaration:

```
- (NSUInteger)insertionPointIndex;
```

Define it in the `DiaryWindowController.m` implementation file:

```
- (NSUInteger)insertionPointIndex {
    return [[[self keyDiaryView] selectedRanges] objectAtIndex:0]
           rangeValue].location;
}
```

In the Cocoa text system, the insertion point is a selection range with a length of 0. When the user clicks the Add Tag button, however, it is entirely possible that a word or phrase may be selected, or even several words or phrases now that the text system supports multiple selection. You must therefore implement a convention defining what the application will consider to be the insertion point and how it will behave when inserting a new tag title. The location of new tag titles is defined in the application specification, so it cannot replace the user's current selection. The insertion point is needed only to know where to begin searching for title markers and tag markers.

In this method, you treat the location of the first selection in the text view's array of selection ranges as the insertion point. According to the *NSTextView Class Reference*, the Cocoa text system guarantees that `-selectedRanges` always returns an array having at least one element, so you do not have to test for these conditions.

8. At last, you are ready to write the `-addTag:` action method. This affects the MVC view, so it belongs in the `DiaryWindowController` class. You will find it in the downloadable project file for Recipe 4.

The `-addTag:` action method begins by defining three local variables, `keyView`, `storage`, and `tagString`, following the pattern of `-addEntry:`. The `tagString`

variable uses the `DIARY_TAG_MARKER` macro, which you defined at the beginning of this step as the Unicode WHITE FLAG character, with code point 0x2690.

The method next calls the `-currentEntryTagRangeForIndex:` method you just finished writing, passing to it the current insertion point. If its `location` member has the value `NSNotFound`, the method returns without doing anything because this means the insertion point is currently in an untitled preface. The application specification dictates that an untitled preface cannot hold a valid tag list. You don't really want an action method to do nothing when its button is clicked or its menu item is chosen, because this will confuse the user. You will shortly arrange for the Add Tag button to be disabled in this circumstance. Nevertheless, you should leave this line in the method as a backstop.

The next line tests the `length` member of the range returned by the `-currentEntryTagRangeForIndex:` method. If it is 0, there is no existing tag marker in the current entry and you should insert a new, empty tag title at `location`. You do this using the same techniques you used in the `-addEntry:` action method in Step 2, complete with undo support and generation of appropriate notifications. The Edit menu will hold Undo Add Tag and Redo Add Tag menu items. If the length is not 0, then there is already a tag title in the current entry and you don't have to add one.

Finally, whether you're adding a new tag title or one already exists in the current entry, the method scrolls the text view having keyboard focus to the new or existing tag title and places the insertion pointer appropriately so that the user can begin typing a new tag.

9. As with every new action method, you must remember to connect it up in Interface Builder. Open the `DiaryWindow` nib file and Control-drag from the Add Tag button in the diary window to the First Responder proxy in the nib file's window. In the HUD, choose the `addTag:` action.
10. You should build and run the application now to test the Add Entry and Add Tag buttons in combination. Open the Chef's Diary window and click the Add Entry button. Immediately after that, click the Add Tag button. In the line following the new entry's title, you see a white flag character and the word *Tags:*, and the insertion point is located just after that so you can type a tag. Type **dessert**, **appetizer** or something similar. Then press the Return key and write up a real or imagined culinary experience, using as many paragraphs as you like. Then click Add Entry and Add Tag again and type some more. Now experiment. Click in the middle of one of the entries and click Add Entry. A new entry appears at the end of the diary, and if the end was scrolled out of view, it scrolls into view. Click again in the middle of any entry and click Add Tag. If that entry already has a tag title, the insertion point moves to its end to let you type another tag.

Now click at the beginning of the diary, before the first entry's title marker. Type some text, creating an untitled preface to the diary, followed by Return, and click in the new text. Notice that the Add Tag button remains enabled, but when you click it, nothing happens. Now delete all the text from the Chef's Diary window. The Add Tag button is still enabled. Click Add Tag again, and still nothing happens. You will arrange to disable the Add Tag button in either of these circumstances in Step 4.

Step 4: Validate the Add Tag Push Button

Good user interface design requires, among other things, that you take pains to avoid confusing the user. One potential source of confusion is user interface elements that look as though they're available but don't do anything. For this reason, many controls and other user interface elements can be enabled or disabled, with distinctive visual differences that users have come to understand. These include toolbar items, menu items, and all kinds of buttons. You should make sure that your application's eligible UI elements are disabled whenever the state of the application is such that they aren't functional.

In this step, you disable the Add Tag button when the insertion point in the active text view is positioned in an area that has no entry marker preceding it. Either the text view is empty, or the insertion point is located in an untitled preface. The application specification forbids tag titles in these circumstances, so the Add Tag button should be disabled. In all other circumstances, it should be enabled.

You enable and disable a button by sending it the `-setEnabled:` message with a parameter value of YES to enable it or NO to disable it. The straightforward way of doing this is to declare an outlet for the control, write accessor methods to get the outlet and perhaps set it, and connect the outlet in Interface Builder. Then you can send the outlet a `setEnabled:` message at the right time.

It is common practice to call the `-setEnabled:` method multiple times in a custom method that validates every user interface item in a window. The custom method is typically named something like `-updateWindow`. An application might call the `-updateWindow` method from several other methods, perhaps once from `-awakeFromNib` or `-windowDidLoad` to validate controls when the window first opens, and then again from any method that is called when an event occurs that requires the window's controls to be validated again.

Sending a message directly to a connected outlet is fast, but this technique requires you to do a significant amount of work every time you add a new control to the application. In a complex application, you may end up declaring, implementing, and connecting dozens or even hundreds of outlets.

Cocoa offers a more generalized way to perform user interface validation, the `NSUserInterfaceValidations` protocol, which you use in this step. It saves you the trouble of having to declare and connect all those outlets. It also has the advantage of integrating automatically with menu item validation, saving you even more effort. In this context, *validation* refers to the control's enabled or disabled state. The term is used in other contexts to mean something different, such as determining whether the content of a text field or the value of a date picker is valid and should be shown or hidden.

Before going any further, read the “Protocols” sidebar to understand what a protocol is.

Protocols

Protocols are a feature of the Objective-C programming language. Creating protocols of your own is optional, but they play an important role in the Cocoa frameworks. In Mac OS X 10.6 Snow Leopard, the number of formal protocols expanded greatly, because now all delegate methods are declared as formal protocols. Prior to Snow Leopard, they were declared as informal protocols using another Objective-C language feature, categories. According to the *Application Kit Framework Reference* and the *Foundation Framework Reference*, at this writing there are 64 protocols in the AppKit and 29 in Foundation.

For Cocoa beginners, it is almost as easy to overlook protocols as it is to overlook the even more deeply hidden global Cocoa functions, types, and constants. You should make a point of becoming familiar with protocols. Read the Protocols chapter of the *Objective-C Programming Language* for complete details.

Protocols declare methods that aren't associated with any particular class. Thus, they fall outside the normal class hierarchy. You can use a protocol many times in many different classes, without regard to their inheritance structure, and any class can adopt multiple unrelated protocols. For this reason, you can think of protocols as forming another network of data types, separate from the class hierarchy. Classes can be grouped conceptually both according to their inheritance structure and according to the protocols they adopt. Objective-C offers means to ascertain whether any class uses a particular protocol and to identify all of the classes that use it. There are even protocol objects that Cocoa can pass as arguments to methods.

(continues on next page)

One use for protocols is to implement something very like the multiple inheritance capability that exists in some other object-oriented languages. If you examine the `NSCopying` protocol in Foundation, for example, you discover that it declares a method that must be implemented by any object that supports copying. A great many Cocoa classes adopt the `NSCopying` protocol. By doing so, they enable the compiler to ensure that objects you attempt to copy implement the required method. Other protocols declare larger sets of required methods. If declared in subclasses or categories, these could be used only within a class hierarchy; declaring them as protocols means they can be used anywhere.

There are two kinds of protocols, *formal* and *informal*. Neither has an implementation part, because a protocol's clients implement the protocol's declared methods.

A formal protocol is declared using the `@protocol` directive. A client class or category adopts it by importing its header file and including its protocol name, or a comma-delimited list of protocol names, in angle brackets at the end of the `@interface` directive. Adopting a formal protocol in one of your classes constitutes a guarantee that your class implements all of the protocol's required methods, and the compiler complains if it doesn't. Beginning with Objective-C 2.0, formal protocols can declare methods as optional using the `@optional` directive. By default, methods are required, and they can be expressly declared as required using the `@required` directive. A class or category adopting a formal protocol does not include declarations of the protocol's methods in its interface, because the adoption of the protocol in the `@interface` directive guarantees that all of those that are required are implemented.

A formal protocol, like classes and other exported symbols, should have a two- or three-letter uppercase prefix to ensure that it does not conflict with a protocol having the same name in frameworks you import from Apple or a third party. Protocols have a namespace of their own separate from the global namespace used by classes, but there is still a risk of namespace collision. See the "Code Naming Basics" section of Apple's *Coding Guidelines for Cocoa* for details.

An informal protocol is typically declared as a category on `NSObject` without an implementation part. To use the methods that it declares, a client class or category does nothing more than implement those of its declared methods that it needs, just as it would implement the methods of any category. There is no special type checking to verify that you have done so correctly. Indeed, it is the lack of type checking for informal protocols that led Apple to start declaring delegate methods in formal protocols.

The basic concept underlying the `NSUserInterfaceValidations` protocol is that a single method in a controller class holds all the code that is required to decide whether to enable or disable all of the UI elements within the controller's jurisdiction. That method is the sole method declared in the `NSUserInterfaceValidations` protocol, `-validateUserInterfaceItem:`, which you will implement. It is important to name it `-validateUserInterfaceItem:` and to declare that your controller conforms to the protocol so that other parts of Cocoa are aware of it.

The parameter to the `-validateUserInterfaceItem:` protocol method is an eligible user interface item, such as a toolbar item, a menu item, or a validated control. In the body of the method, you ascertain the identity of the specific item by its action method (or, rarely, by its tag). The item must conform to the related `NSValidatedUserInterfaceItem` protocol or to a custom validation protocol that you declare. Technically, conformity to the `NSValidatedUserInterfaceItem` protocol guarantees that the item implements the `-action` and `-tag` methods. In practice, conformity also means that the item implements a `-setEnabled:` method and is meant to be validated using the techniques described here. You then specify the conditions under which the item should be enabled or disabled. You arrange for the method to return `YES` if the current state of the application is such that the item should be enabled, or `NO` if it should be disabled. You don't have to declare or connect outlets to the items because each item in succession is passed into the protocol method.

For the Add Tag button, for example, your implementation of the `-validateUserInterfaceItem:` method first ascertains whether the item's action is the `addTag:` action. If it is, the method tests whether the insertion point in the diary window's active text view is within a diary entry. If so, the method returns `YES`; if not, it returns `NO`. If the item's action is not the `addTag:` action, the method returns `YES` so that all the other items in the window are enabled. You have to do this because the protocol method may be called on other items in the window that you don't intend to validate.

The trick is to know when and how to call the `-validateUserInterfaceItem:` protocol method. Cocoa automatically validates two kinds of user interface items, menu items and toolbar items, if you implement `-validateUserInterfaceItem:` or the more specialized `-validateMenuItem:` and `-validateToolbarItem:` methods. Cocoa handles controls such as buttons differently. You still have to implement `-validateUserInterfaceItem:` or a more specialized validation method, but Cocoa does not call them automatically. You have to call them yourself whenever changes to the window require validation of controls.

To understand how validation works, consider first `-validateMenuItem:` and `-validateToolbarItem:`. Cocoa calls these methods automatically if you implement them, much like the automatic invocation of delegate methods that you have implemented. Menu items are validated when an `NSMenu` object's

`-update` method is called, and toolbar items are validated when an `NSToolbar` object's `-validateVisibleItems` method or an `NSToolbarItem` object's `-validate` method is called. In the case of menu items, validation occurs whenever the user opens a menu and thus triggers its `-update` method. In the case of toolbar items, validation occurs when an `NSWindow` object's `-update` method calls the toolbar's `-validateVisibleItems` method, which happens in every iteration of the run loop. You can override these methods to make them more efficient if your validation code takes too much time. You can call `NSMenu`'s `-setAutoenablesItems` and `NSToolbarItem`'s `-setAutovalidates:` method to turn automatic validation of menu items and toolbar items on or off. The final point to understand is that menu item and toolbar item validation falls back on `-validateUserInterfaceItem:` if you implement it and don't implement `-validateMenuItem:` or `-validateToolbarItem:`. This is important because it allows you to put validation code in a single method, `-validateUserInterfaceItem:`, if your application has controls and menu items that respond to the same action message, as most applications do. You will do this in Recipe 5 by adding an Add Tag menu item and others duplicating the actions of some of the controls, which will be validated by the same `-validateUserInterfaceItem:` protocol method you implement here.

To use the `-validateUserInterfaceItem:` protocol method with controls, the end result is the same, but the way you set it up is a little different. You have to supply some of the support for validated controls, whereas Cocoa provides this support for you in the case of menu items and toolbar items. Your application not only must implement the `-validateUserInterfaceItem:` protocol method, but also must call the protocol method explicitly for controls. Alternatively, your application can implement and call a more specialized validation method analogous to `-validateMenuItem:` and `-validateToolbarItem:`, which you might name something like `-validateControl:`. The documented way to do this follows the same pattern that is built into Cocoa for menu items and toolbar items. You declare two custom protocols; you subclass the controls that are to be validated, so that they conform to one of the protocols and implement its required validation method; and finally, you call the controls' validation methods from your controller whenever the user changes the state of the window. You implement the documented technique in Vermont Recipes. See the “Implementing a Validated Item” section of *User Interface Validation* for details.

Normally, you validate user interface items every time the control's window is updated. You could do this, for example, in your implementation of `NSWindow`'s `-windowDidUpdate:` delegate method, which the system calls automatically every time the window updates—that is, once in every iteration of the run loop. If the application's logic is clear or speed is an issue, you can be more discriminating and validate controls only in response to specific relevant events, but you shouldn't go that route until profiling of your finished application demonstrates a real performance issue.

Here, you start by declaring two protocols, `VRValidatedControl` and `VRControlValidations`. Protocol names must be unique. *The Objective-C Programming Language* indicates that they do not have global visibility and live in their own namespace, unlike classes. Nevertheless, you should name them with a prefix consisting of two or three uppercase letters to minimize the risk of namespace collision with frameworks you import from Apple or a third party. Here, you use `VR`, for Vermont Recipes. The `VRValidatedControl` protocol declares a single method, `-validate`. You label it `@required`, because every validated control must implement this method. The `VRControlValidations` protocol declares the optional method `-validateControl:`. Then you implement a subclass of `NSButton` named `ValidatedDiaryButton`, and you declare that it conforms to the `VRValidatedControl` protocol. Conformity means that the button implements the `-validate` method. You implement `-validate` to call the controller's `-validateControl:` method, if it implements one, and if not, to fall back on the controller's `-validateUserInterfaceItem:` protocol method. Finally, in the window controller, you implement `-validateUserInterfaceItem:` to perform the tests needed to decide whether the control should be enabled or disabled, and you call it on every validated control when the window updates, once in every iteration of the run loop.

The controller could implement `-validateControl:` instead of `-validateUserInterfaceItem:`, but for the Chef's Diary window you want to have a number of menu items that perform the same actions as the validated buttons.

1. Start by implementing the protocol method, `-validateUserInterfaceItem:`. Enter it in the `DiaryWindowController.m` source file, following the existing `-windowDidLoad` method. You don't have to declare it in the header file because it is declared in the `NSUserInterfaceValidations` protocol.

```
- (BOOL)validateUserInterfaceItem:
    (id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if (action == @selector(addTag:)) {
        return ([[self document] currentEntryTitleRangeForIndex:
                [self insertionPointIndex]].location != NSNotFound);
    }
    return YES;
}
```

You will arrange shortly to call this method in a loop, once for every view in the window that conforms to the `VRValidatedControl` protocol. Only some of the controls will conform to it. Every time the method is called, a reference to a particular validated user interface item is passed to it in the `item` parameter. The type of the parameter is `id` to allow any kind of control to be validated. In Step 6, you will in fact validate the diary window's date picker, which is not a button.

The item must conform to the `NSValidatedUserInterfaceItem` protocol because of the reference to the protocol in angle brackets as part of the type declaration. Every item that conforms to the `VRValidatedControl` protocol meets this condition, because you will shortly declare the protocol to inherit from the `NSValidatedUserInterfaceItem` protocol.

You test the current `item` value to determine whether its `action` is the `addTag:` action. If it is, then this must be the Add Tag button (or an Add Tag menu item that sends the same action), so you return a Boolean value of `YES` or `NO` depending on whether the `-currentEntryTitleRangeForIndex:` method returns a range whose `location` member is `NSNotFound`. If this item is not the Add Tag button, the method falls through the `if` clause and returns `YES` in all other cases, because for now, at least, all other buttons in the window should always be enabled. You will shortly add additional `if` clauses to test for other validated buttons. A typical `-validateUserInterfaceItem:` method might have many `if` clauses.

This is not the first time you have encountered an Objective-C *selector*, but it wasn't previously singled out as a distinct feature of the language. You have seen selectors every time you connected a user interface element in Interface Builder and chose an action. A selector is a language element of type `SEL`. Look up the `-action` method in the *NSControl Class Reference*, and you see that it returns a value of type `SEL`. To test whether this action is a particular selector, you use the `@selector` compiler directive, passing in the selector, or action, you're interested in. As you see here, a selector is simply the method's signature, including all colons marking parameters and the parameter labels, which uniquely identifies this action. Note that the selector is not a string object. Should you ever want to display a selector, perhaps in an `NSLog()` function call for debugging purposes, use the Cocoa `NSStringFromSelector()` function.

2. Because the `DiaryWindowController` class now implements the `-validateUserInterfaceItem:` protocol method, you should change its header file to advertise this fact. In `DiaryWindowController.h`, change the `@interface` directive to this:

```
@interface DiaryWindowController :  
    NSWindowController <NSUserInterfaceValidations> {  
}
```

As you see, you declare conformity to a formal protocol by placing its name in angle brackets at the end of the `@interface` directive. This is known as the *protocol list*. To declare conformity with multiple protocols, separate them with commas in the protocol list.

3. Next, implement the method that calls the `-validateUserInterfaceItem:` protocol method. In Vermont Recipes, this is a custom `-updateWindow` method. Place it after the existing `-windowDidLoad` method.

In the `DiaryWindowController.h` header file, declare it like this:

```
- (void)updateWindow;
```

In the `DiaryWindowController.m` source file, implement it like this:

```
- (void)updateWindow {
    for (id thisView in [[[self window] contentView] subviews]) {
        if ([thisView conformsToProtocol:
            @protocol(VRValidatedControl)]) {
            [thisView validate];
        }
    }
}
```

The method loops through all the subviews in the diary window's content view using the new fast enumeration syntax in Objective-C 2.0. In each iteration of the loop, it tests whether the current view conforms to the `VRValidatedControl` protocol. A protocol is specified for purposes of the `-conformsToProtocol:` method using the `@protocol` directive, which is similar to the `@selector` directive. Most of the views in the window do not conform, including the Add Entry button. They are therefore skipped.

When the Set Tag button is checked in the loop, it meets the test. It conforms to the protocol because you are about to make it so. The method therefore calls this control's `-validate` protocol method, which does the work of enabling or disabling the control.

4. Next, arrange to call the new `-updateWindow` method at the appropriate times—namely, whenever the window updates. To do this, implement `NSWindow`'s `-windowDidUpdate:` delegate method immediately before the `-updateWindow` method, like this:

```
- (void)windowDidUpdate:(NSNotification *)notification {
    [self updateWindow];
}
```

It may seem wasteful to implement a separate `-updateWindow` method when its body could have been placed directly in the `-windowDidUpdate:` delegate method. Over time, however, you may find that you sometimes need to call an `-updateWindow` method from other methods as well as from the delegate method, so it can be convenient to make it a separate method now.

5. None of this works unless the Set Tag button actually does conform to the `VRValidatedControl` protocol. Objective-C tests conformity to formal protocols by checking whether a class's `@interface` directive includes the protocol in angle brackets in the *protocol list*. Cocoa does not check to see whether the class actually responds to the methods required by the protocol; it is the programmer's responsibility to implement them if the class declares that it conforms.

To make validation work here, you must therefore declare and implement a subclass of `NSButton` and declare that the subclass conforms to the `VRValidatedControl` protocol. The new subclass—call it `ValidatedDiaryButton`—has to implement only one method, namely, the `-validate` protocol method. It inherits the other methods it needs, `-action` and `-setEnabled:`, from `NSButton`.

The subclass is relevant only to the diary window, so it is appropriate to declare and implement it in the `DiaryWindowController` files. At the end of the `DiaryWindowController.h` header file, after the `@end` directive, declare it like this:

```
@interface ValidatedDiaryButton : NSButton <VRValidatedControl> {  
}  
@end
```

It doesn't declare the `-validate` method because it declares conformity to the `VRValidatedControl` protocol, which does declare it, thus guaranteeing that it implements that method.

At the end of the `DiaryWindowController.m` implementation file, after the `@end` directive, implement it like this:

```
@implementation ValidatedDiaryButton  
- (void)validate {  
    id validator =  
        [NSApp targetForAction:[self action]  
        to:[self target] from:self];  
    if ((validator == nil)  
        || ![validator respondsToSelector:[self action]]) {  
        [self setEnabled:NO];  
    } else if ([validator respondsToSelector:  
        @selector(validateControl:)]) {  
        [self setEnabled:[validator validateControl:self]];  
    } else if ([validator respondsToSelector:  
        @selector(validateUserInterfaceItem:)]) {  
        [self setEnabled:[validator validateUserInterfaceItem:self]];  
    } else {  
        [self setEnabled:YES];  
    }  
}  
@end
```


If you decide later to add any buttons to the window that require validation and are subclasses of `NSButton`, such as `NSPopUpButton`, you will have to declare them as validated subclasses as well. You can also add other controls in the same way. In Steps 6 and 7, for example, you will validate the diary window's date picker and search field in this manner, although they are not subclasses of `NSButton`.

6. There is one more thing to do before you declare the protocols that lie behind this validation technique. At this point, the Set Tag button still thinks its class is `NSButton`, which does not conform to the protocol. You have to change its class to `ValidatedDiaryButton`.

First, build the application. Then, in Interface Builder, select the Set Tag button in the diary window. In the Button Information inspector, choose `ValidatedDiaryButton` from the Class pop-up menu. Save the nib file, and build the application again.

7. Finally, you're ready to declare the protocols. This may come as an anticlimax after all that, because there is nothing to it.

Near the end of the `DiaryWindowController.h` header file, just above the new `-ValidatedDiaryButton` declaration, insert these two protocol declarations:

```
@protocol VRValidatedControl <NSValidatedUserInterfaceItem>
@required
- (void)validate;
@end

@protocol VRControlValidations
@optional
- (BOOL)validateControl:(id <VRValidatedControl>)item;
@end
```

8. Run the application and test the Add Tag button. When you first open a new diary window, the button is disabled, as it should be, because there is no entry title before the insertion point. Now type a few characters, press the Return key, and click Add Entry. The Add Tag button immediately becomes enabled because the insertion point is now positioned after the new diary entry's title marker and is thus in the current entry. Click Add Tag, and a new tag list appears. Move the insertion point before the entry's title marker, using the mouse button or the arrow keys on the keyboard, and the Add Tag button immediately becomes disabled.

Step 5: Implement and Validate the Navigation Buttons

Approach the four navigation buttons in the same way you did the Add Tag button. First write methods that return the range of the target entry's title. Then write a separate action method for each to scroll the title into view and select it. Finally, subclass the buttons to take advantage of the validation protocols you have just written.

You already implemented two of the range methods underlying the navigation buttons in Step 3, `-currentEntryTitleRangeForIndex:` and `-nextEntryTitleRangeForIndex:`. Now implement the other three, `-firstEntryTitleRange`, `-lastEntryTitleRange`, and `-previousEntryTitleRangeForIndex:`, as well as all four of the action methods.

1. Declare and implement the three new range methods in the `DiaryDocument` class. You will find them in the downloadable project file for Recipe 4.

These three methods follow the same strategy as `-currentEntryTitleRangeForIndex:`. The first two define the entire text storage as the search range, so they don't require an index parameter. The `-firstEntryTitleRange` method searches forward from the beginning of the diary, while `-lastEntryTitleRange` searches backward from the end. The first entry marker character they find marks the location of the targeted entry title. The last method, `-previousEntryTitleRangeForIndex:`, finds the current entry by calling `-currentEntryTitleRange` and then searching backward for the previous entry marker.

2. Next, write the four navigation action methods. Name each of them with *goTo* as the first part of the method name, on the model of several methods in AppKit's `PDFView` class. All of them select an entry's title using the corresponding range method to get the target range, and then scroll it into view. They also give keyboard focus to the appropriate text view, in case the search field had focus when the user clicked the navigation button.

As action methods, the navigation methods belong in `DiaryWindowController`. You will find them in the downloadable project file for Recipe 4.

3. Connect the action methods.

In the `DiaryWindow` nib file, Control-drag from the top-left navigation button in the diary window to the First Responder proxy in the nib file's window; then choose the `goToFirstEntry:` action in the Received Actions section of the HUD. Alternatively, Control-click (right-click) the top-left navigation button, and then drag from the marker beside the `goToFirstEntry:` action in the Sent Actions section of the HUD to the First Responder proxy; or Control-click the First Responder proxy and choose the `goToFirstEntry:` action in the Received Actions section of the HUD.

Do the same thing with the other three navigation buttons, connecting the bottom-left button to the `goToLastEntry:` action, the top-right button to the `goToPreviousEntry:` action, and the bottom-right button to the `goToNextEntry:` action.

4. The four navigation buttons work now, but they ought to be disabled when there is nothing for them to do. All of the buttons should be disabled when there are no entries in the diary, and the top-right and bottom-right buttons should also be disabled when there is no entry before or after the current entry. Otherwise, it is always useful to scroll the targeted entry's title into view and, even if it is already scrolled into view, to select it so that it catches the user's eye.

You already set up the validation machinery for the diary window in Step 4. You have to do only two additional things to bring the navigation buttons under its control.

First, set the class of the four navigation buttons to `ValidatedDiaryButton`. To do this, select each of them in turn, and in the Button Identity inspector, choose `ValidatedDiaryButton` in the Class pop-up menu.

Second, add several clauses to the `-validateUserInterfaceItem:` protocol method you wrote in Step 4. This is the revised method in full:

```
- (BOOL)validateUserInterfaceItem:
    (id <VRValidatedControl>)item {
    SEL action = [item action];
    if (action == @selector(addTag:)) {
        return ([[self document] currentEntryTitleRangeForIndex:
                [self insertionPointIndex]].location != NSNotFound);
    } else if (action == @selector(goToFirstEntry:)) {
        return [[self document] firstEntryTitleRange].location
            != NSNotFound;
    } else if (action == @selector(goToLastEntry:)) {
        return [[self document] lastEntryTitleRange].location
            != NSNotFound;
    } else if (action == @selector(goToPreviousEntry:)) {
        return [[self document] previousEntryTitleRangeForIndex:
                [self insertionPointIndex]].location != NSNotFound;
    } else if (action == @selector(goToNextEntry:)) {
        return [[self document] nextEntryTitleRangeForIndex:
                [self insertionPointIndex]].location != NSNotFound;
    }
    return YES;
}
```

If the button being checked is not the button connected to the `addTag:` action, the method checks whether it is the button connected to the `goToFirstEntry:` action. If it isn't that button, it continues down the chained `else if` clauses looking for each of the other navigation buttons. When it determines that one of the navigation buttons is currently being validated, it searches for that button's target entry title range. Finally, it returns the Boolean value `YES` if the target range is found and otherwise `NO`. As in Step 4, it returns `YES` for all other buttons to ensure that they are enabled.

5. Run the application and test the navigation buttons. Add several entries to the diary using the Add Entry button. Then click the navigation buttons in a variety of ways and watch what happens. Verify through observation that they are enabled and disabled as designed.

Step 6: Implement and Validate the Date Picker

A Cocoa date picker control serves two useful purposes in the Vermont Recipes application.

First, the date picker acts as a more discriminating `goTo` control than the four navigation buttons. The navigation buttons are limited to the first, last, previous, and next entries. Even the most experienced chef will quickly generate more diary entries than that. Since each entry is titled with the date when it was created, a date picker is a perfect control to navigate to a specific entry. In this step, you code the date picker to navigate to the first, or oldest, entry that is equal to or more recent than the date entered in the date picker. In other words, the date picker's setting defines the oldest entry that the user wants to select. In the special case where there is no entry more recent than the date picker's setting, it navigates to the most recent entry in the diary.

Second, the date picker acts as a title for the current entry, so the user can see at a glance which entry is currently active even if its title has scrolled off the top of the window. It is updated automatically, as the user navigates through the diary, to display the date of the current entry. It should do this only when the user is not using the date picker itself to navigate. When the date picker is in use, it shows the date being set by the user. It reverts to displaying the current entry's date when the user returns keyboard focus to the text view.

When the date picker is not in use and there is no current entry—that is, when the text view is empty or the insertion point is in an untitled preface—the date picker defaults to displaying the current date.

Implement the date picker's action method first, and then turn to its role as title for the current diary entry.

1. Cocoa's `NSDatePicker` class inherits from `NSControl`, so it responds to `NSControl`'s `-action` method. Start by implementing a `-goToDatedEntry:` action method, on the model of the other `goTo` methods. It will have to call a new range method, which you will write shortly.

In the `DiaryWindowController.h` header file, declare the action method after the last of the other `goTo` action methods:

```
- (IBAction)goToDatedEntry:(id)sender;
```

In the `DiaryWindowController.m` source file, define it like this after the last `goTo...` method:

```
- (IBAction)goToDatedEntry:(id)sender {
    NSTextView *keyView = [self keyDiaryView];
    NSRange targetRange =
        [self firstEntryTitleRangeAtOrAfterDate:[sender dateValue]];
    if (targetRange.location != NSNotFound) {
        [keyView scrollRangeToVisible:targetRange];
        [keyView setSelectedRange:targetRange];
    }
}
```

Only two features distinguish this from the other `goTo` action methods: It calls a different range method, `-firstEntryTitleRangeAtOrAfterDate:`, which you have yet to write, and it does not return keyboard focus to the text view.

The latter point is important for usability. The date picker should remain in control until the user explicitly clicks in the text view. This allows the user to continue changing the settings on the date picker without interruption.

2. Write the `-firstEntryTitleRangeAtOrAfterDate:` range method. Like the other range methods, it belongs in `DiaryDocument`.

In the `DiaryDocument.h` header file, declare it following `-nextEntryTitleRangeForIndex:` as follows:

```
- (NSRange)firstEntryTitleRangeAtOrAfterDate:(NSDate *)date;
```

In the `DiaryDocument.m` implementation file, declare it in the same relative location, like this:

```
- (NSRange)firstEntryTitleRangeAtOrAfterDate:(NSDate *)targetDate {
    NSDate *date;
    NSRange tempRange;
```

(code continues on next page)

```

NSRange returnRange = [self firstEntryTitleRange];
if (returnRange.location == NSNotFound)
    return NSRange(NSNotFound, 0);
do {
    date = [self dateFromEntryTitleRange:returnRange];
    tempRange = [self nextEntryTitleRangeForIndex:
        returnRange.location + returnRange.length];
    if (tempRange.location == NSNotFound) break;
    if ((date == nil)
        || ([date timeIntervalSinceDate:targetDate] < 0))
        returnRange = tempRange;
} while ((date == nil)
    || ([date timeIntervalSinceDate:targetDate] < 0));
return returnRange;
}

```

The method starts by declaring the `date`, `tempRange`, and `returnRange` local variables. It calls `-firstEntryTitleRange` to set `returnRange` to the first entry title in the diary, returning a range with location `NSNotFound` if the diary contains no entries. It then enters the loop, calling `-nextEntryTitleRangeForIndex:` to look for each successive entry title until it runs out of titles or finds one that is more recent than the target date. It calls a method that you have yet to write, `-dateFromEntryTitleRange:`, to convert each title to an `NSDate` object if possible. It ignores entry titles that are malformed, indicated by a `nil` date. When it runs out of entry titles or finds one that is more recent than the target date, it returns the range with the last qualifying date.

3. Write the `-dateFromEntryTitleRange:` method. Once you've done this, the support methods for the `-goToDatedEntry:` action method will be complete.

In the `DiaryDocument.h` header file, declare it immediately following the `-firstEntryTitleRangeAtOrAfterDate:` method:

```
- (NSDate *)dateFromEntryTitleRange:(NSRange)range;
```

Define it as follows in the same relative location in the `DiaryDocument.m` implementation file:

```

- (NSDate *)dateFromEntryTitleRange:(NSRange)range {
    if ((range.location == NSNotFound)
        || (range.length < 2)) return nil;
    range.location += 2;
    range.length -= 2;
    NSString *dateString = [[[self diaryView] textStorage]
        string] substringWithRange:range];
}

```



```

NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
[formatter setDateStyle:NSDateFormatterLongStyle];
[formatter setTimeStyle:NSDateFormatterLongStyle];
NSDate *date = [formatter dateFromString:dateString];
[formatter release];
return date;
}

```

This method returns a date derived from the title range in the range parameter. It returns `nil` if the incoming range value is an `NSNotFound` range, or if its length is less than 2. The length test is based on the fact that all entry titles begin with the entry marker character and a space; without those, the title can't hold a date. The test is necessary so that the immediately following range manipulations do not cause an error if the user has deleted all of the title string except the initial title marker.

The method then adjusts the incoming range to eliminate from consideration the leading title marker and space.

The rest of the title should be a human-readable date string using `NSDateFormatter`'s `NSDateFormatterLongStyle`, but the user might have edited it so that it no longer represents a date that Cocoa can interpret. Fortunately, `NSDateFormatter`'s `-dateFromString:` method returns `nil` if it is unable to read the string. After allocating and initializing a new formatter and setting its date and time styles, the method calls `-dateFromString:`, releases the formatter, and returns the date as an `NSDate` object.

4. The `-goToDatedEntry:` action method will now work, once you connect it up in Interface Builder. However, you should take steps to disable the date picker when there are no entries in the diary. First, add this clause at the end of the `-validateUserInterfaceItem:` implementation in the `DiaryWindowController.m` source file:

```

} else if (action == @selector(goToDatedEntry:)) {
    return [[self document] firstEntryTitleRange].location != NSNotFound;
}

```

You call the existing `-firstEntryTitleRange` method for the test range because if there is no first entry, there is no entry at all.

Next, declare a subclass of `NSDatePicker` called `ValidatedDiaryDatePicker`. This should be the same as the `ValidatedDiaryButton` subclass at the end of the `DiaryWindowController` files except that it inherits from `NSDatePicker`, not from `NSButton`. Declare it like this:

```

@interface ValidatedDiaryDatePicker :
    NSDatePicker <VRValidatedControl> {
@end

```

Implement it exactly the same as the ValidatedDiaryButton's `-validate` method:

`@implementation ValidatedDiaryDatePicker`

```
- (void)validate {
    id validator =
        [NSApp targetForAction:[self action]
         to:[self target] from:self];
    if ((validator == nil)
        || ![validator respondsToSelector:[self action]]) {
        [self setEnabled:NO];
    } else if ([validator respondsToSelector:
        @selector(validateControl:)]) {
        [self setEnabled:[validator validateControl:self]];
    } else if ([validator respondsToSelector:
        @selector(validateUserInterfaceItem:)]) {
        [self setEnabled:[validator validateUserInterfaceItem:self]];
    } else {
        [self setEnabled:YES];
    }
}

@end
```

You must also turn the date picker into a ValidatedDiaryDatePicker. Select the date picker in the diary window in Interface Builder, and then in the Validated Diary Date Picker identity inspector, choose ValidatedDiaryDatePicker as its class.

5. The second task in this step is to tell the date picker what date to display while the user is editing or navigating in the diary. To do this, you must first write an outlet for the date picker to act as the receiver for the message.

In the DiaryWindowController.h header file, add this declaration at the end of the instance variable declarations between the braces of the `@interface` directive:

```
IBOutlet NSDatePicker *datePicker;
```

Add this accessor method after the `-otherDiaryView` accessor:

```
- (NSDatePicker *)datePicker;
```

In the DiaryWindowController.m implementation file, add this implementation after the `-otherDiaryView` accessor:

```
- (NSDatePicker *)datePicker {
    return [[datePicker retain] autorelease];
}
```

Build the project, and then, in Interface Builder, Control-drag from the File's Owner proxy to the date picker in the diary window and choose `datePicker` from the Outlets section of the HUD.

6. Next, write a method to update the date picker's displayed value based on the range of the current entry's title. Call it `-updateDatePickerValue`. In the `DiaryWindowController.h` header file, add the declaration after `-updateWindow`, as follows:

- `(void)updateDatePickerValue;`

Define it like this:

```
- (void)updateDatePickerValue {
    NSRange range = [self currentEntryTitleRangeForIndex:
        [self insertionPointIndex]];
    if (range.location == NSNotFound) {
        [[self datePicker] setDateValue:[NSDate date]];
    } else {
        [[self datePicker] setDateValue:
            [[self document] dateFromEntryTitleRange:range]];
    }
}
```

If there is no current entry or the insertion point is in an untitled preface, the method sets the displayed value of the date picker to the current date and time, using `NSDatePicker`'s `-setDateValue:` method. This happens both when a new, empty window is opened and when the user moves the insertion point into the preface.

If there is a current entry, the method sets the date picker to display its title in the form of a date. If the date value of the current entry's title is `nil` because the user has edited it, nothing happens because `-setDateValue:` does not do anything with a `nil` parameter value.

7. Finally, call the `-updateDatePickerValue` method every time the window updates. Add this statement at the end of the existing `-updateWindow` method implementation:

```
if (![self window] firstResponder
    isKindOfClass:[NSDatePicker class])) {
    [self updateDatePickerValue];
}
```

It is necessary to verify that the date picker does not currently have keyboard focus, because the date picker should not be updated automatically while the user is using it to set a date value.

8. Open the `DiaryWindow.nib` file in Interface Builder. Control-drag from the date picker in the diary window to the First Responder proxy in the `DiaryWindow.nib` file's window, and then select the `goToDatedEntry:` action in the Received Actions section of the HUD. The date picker is now connected. Changing any of its values, such as the seconds, while the application is running executes the action method. This can have a highly interactive feeling in the date picker because, if you hold down the increment or decrement button to change the value continuously, the selection changes every time the value passes a threshold.
9. Run the application and test the date picker. This requires patience, because it works to best advantage if a little time separates each entry. Taking your time, add three or four entries separated by at least several seconds. Then click in different entries at random, and confirm that the date picker changes to reflect the date and time of the current entry's title. Then click to select the seconds cell in the date picker, and click the increment and decrement buttons to change the time backward and forward. As you do this, watch the selection in the diary's text. It moves from title to title based on the principle that the first entry at or after the date set in the date picker is selected.

Step 7: Implement and Validate the Search Field

The Vermont Recipes application has a general Find command in the Edit menu, courtesy of the Cocoa document-based application template. Try it out. It finds any text in the diary window, whether the text is located in an entry's title, its tag title, or its text.

But that isn't the only kind of text-based search you want for the Vermont Recipes application. You went to a lot of trouble to provide a tag list in the diary window. It is now time to put it to use by enabling the user to search for tags alone.

The idea behind tags is that you tag every entry with words or very short phrases that describe the content of the entry. Examples of interesting tags are *dessert* and *appetizers*. In this step, you implement the search field at the bottom of the diary window so that it finds tags only in tag lists, not in an entry's text, and it selects and highlights the tags so that you can see them while scrolling through the diary. To find every entry that is tagged with the *dessert* tag, for example, type **dessert** in the search field. Assuming that multiple entries have that tag, the first dessert tag scrolls into view and briefly highlights. Repeatedly clicking Return in the search field scrolls to each successive instance of the *dessert* tag and highlights it. All instances remain selected as you scroll up and down using the scroll bar.

1. To start with, add placeholder text to the search field. In Interface Builder, select the search field and go to the Search Field Attributes inspector. Enter **tag** in the Placeholder field. This word will appear dimmed in the search field when it does not have keyboard focus, to remind the user that the search field filters the diary on the basis of tags. When the user clicks in the search field to begin typing a tag, the placeholder text disappears automatically.
2. Write the `-findTag:` action method. For search fields, the action method you write must search for and display the found text.

In the `DiaryWindowController.h` header file, declare the action method immediately following the existing action methods, like this:

```
- (void)findTag:(id)sender;
```

In the corresponding source file, implement it like this:

```
- (IBAction)findTag:(id)sender {
    NSTextView *keyView = [self keyDiaryView];
    NSString *tagString = [sender stringValue];
    NSArray *tagRangeArray = [[self document]
                             foundTagRangeArrayForTag:tagString];
    if ([tagRangeArray count] > 0) {
        [keyView setSelectedRanges:tagRangeArray];
        static NSUInteger highlightIndex = 0;
        static NSUInteger tagStringLength = 0;
        if (tagStringLength != [tagString length]) highlightIndex = 0;
        NSRange highlightRange =
            [[tagRangeArray objectAtIndex:highlightIndex] rangeValue];
        tagStringLength = [tagString length];
        highlightIndex =
            (highlightIndex < [tagRangeArray count] - 1)
            ? highlightIndex + 1 : 0;
        [keyView scrollRangeToVisible:highlightRange];
        [keyView showFindIndicatorForRange:highlightRange];
    }
}
```

The `-findTag:` action method uses many techniques that you have already seen and used for working with text.

It starts by determining which of the two text fields has keyboard focus, using the `-keyDiaryView` method you wrote earlier. Then it reads the search field's text to ascertain the target tag by calling the `-stringValue` method on the `sender` argument, which in this case is the search field. The algorithm it uses is very simple: There are no tag separator characters. Every character you type into the search field is considered to be part of the tag you're looking for, including spaces and punctuation.

The method then sets up a search for all the tags in the text, from beginning to end, looking for the tag marker character and the immediately following tag label, *Tags:*. It looks only in tag lists. To do this, it calls another range method you will write in the `DiaryDocument` class, `-foundTagRangeArrayForTag:`. As you will see in a moment, that method calls two supporting methods you will write shortly, `-firstTagRange` and `-nextTagRangeForIndex:`. The actual search uses `NSString`'s fast and efficient `-rangeOfString:options:range:` method, which you have already encountered. In the search loop, the range of the first found tag in every tag list is added to a mutable array, `tagRangeArray`.

In the final section of the code, the array of tag ranges is used by `NSTextView`'s `-setSelectedRanges:` method, which selects and highlights all of them wherever they appear in the text. Next, one of the selected tags is scrolled into view, and it is then highlighted with a brief, eye-catching animation generated by `NSTextView`'s `-showFindIndicatorForRange:` method.

The final section of the code implements a simple algorithm to control the order in which instances of the same tag are highlighted on successive presses of the Return key. The search field is configured to display found tags as you type each character into the field. When you type the *h* in *ho*, for example, it highlights the first *h* in the first tag containing an *h*. If you then hit the Return key while *h* is still the tag you're searching for, it highlights the first *h* in the second tag containing an *h*. It continues in this fashion until it runs out of found *h* tags, and then it cycles back to the beginning. If, instead of pressing Return after typing *h*, the user types the *o* in *ho*, the method notices that the length of the search text has changed and restarts the cycle. To keep track of the length of the search text and the index of the tag it highlighted the last time the user pressed Return, it uses two static variables, `highlightIndex` and `tagStringLength`. The values of static variables are saved between invocations of the method. It is safe to use static variables here, instead of instance variables, because you will eventually take steps to ensure that there can never be more than one diary window.

3. Now write the `-foundTagRangeArrayForTag:` method. Like the range methods you have written previously, it belongs in the `DiaryDocument` class.

In the `DiaryDocument.h` header file, declare it:

```
- (NSArray *)foundTagRangeArrayForTag:(NSString *)tag;
```

In the `DiaryDocument.m` implementation file, define it:

```
- (NSArray *)foundTagRangeArrayForTag:(NSString *)tag {
    if ([tag length] > 0) {
        NSString *tagLabel = [NSString stringWithFormat:
                               NSLocalizedString(@"%@ Tags: ",
                                                  @"search string for diary document tag string"),
                               [self tagMarker]];
    }
}
```



```

NSMutableDictionary *tagRangeArray = [NSMutableDictionary array];
NSRange searchRange = [self firstTagRange];
NSRange foundRange;
while (searchRange.location != NSNotFound) {
    searchRange.location += [tagLabel length];
    searchRange.length -= [tagLabel length];
    foundRange =
        [[[self diaryDocTextStorage] string]
         rangeOfString:tag options:0 range:searchRange];
    if (foundRange.location != NSNotFound)
        [tagRangeArray addObject:
         [NSValue valueWithRange:foundRange]];
    searchRange = [self nextTagRangeForIndex:
                   searchRange.location + searchRange.length];
}
return [[tagRangeArray copy] autorelease];
}
return nil;
}

```

The method returns `nil` if no tag or an empty tag is provided.

Otherwise, it creates a search string tailored to the design of the application's tag list label. Then it searches repeatedly through every tag list in the diary matching that pattern. Using `NSString`'s now-familiar `-rangeOfString:options:range:` method, it accumulates the ranges of every appearance of the tag in the Chef's Diary's tag lists into the `tagRangeArray` variable and returns it.

4. Now write the two supporting methods required by `-foundTagRangeArrayForTag:`, `-firstTagRange`, and `-nextTagRangeForIndex:`.

In the `DiaryDocument.h` header file, add these two declarations before the `@end` directive at the end of the `DiaryWindowController` class:

```

- (NSRange)firstTagRange;
- (NSRange)nextTagRangeForIndex:(NSUInteger)index;

```

In the `DiaryDocument.m` source file, implement them like this:

```

- (NSRange)firstTagRange {
    if ([[self diaryDocTextStorage] length] == 0)
        return NSRange(NSNotFound, 0);
}

```

(code continues on next page)

```

    NSInteger markerIndex = [[[self diaryDocTextStorage] string]
        rangeOfString:[self tagMarker] options:0
        range:NSMakeRange(0,
            [[[self diaryDocTextStorage] length]])].location;
    return [self rangeOfLineFromMarkerIndex:markerIndex];
}

- (NSRange)nextTagRangeForIndex:(NSInteger)index {
    if ([[self diaryDocTextStorage] length] == 0)
        return NSRange(NSNotFound, 0);
    NSInteger markerIndex = [[[self diaryDocTextStorage] string]
        rangeOfString:[self tagMarker] options:0
        range:NSMakeRange(index,
            [[[self diaryDocTextStorage] length] - index])].location;
    return [self rangeOfLineFromMarkerIndex:markerIndex];
}

```

These are very similar to methods you’ve already written to return entry title ranges, `-firstEntryTitleRange` and `-nextEntryTitleRangeForIndex:`, and they shouldn’t require further explanation.

5. Don’t forget to validate the search field. Set this up exactly the way you set up validation for the date picker, except test whether the diary contains any tag lists. Declare and implement the `ValidatedDiarySearchField` class in `DiaryWindowController` as a subclass of `NSSearchField`, declaring that it conforms to the `VRValidatedControl` protocol. Set this subclass as the search field’s class in Interface Builder. Finally, add a clause at the end of the `-validateUserInterfaceItem:` method, testing for the `-findTag:` action that you just wrote and checking whether the `-firstTagRange` is found.
6. Open the `DiaryWindow` nib file in Interface Builder. Control-drag from the search field in the diary window to the First Responder proxy in the `DiaryWindow` nib file’s window, and then select the `findTag:` action in the Received Actions section of the HUD. The search field is now connected.
7. Run the application and test the search field. To start, you have to create a few entries and add tags to them, using the Add Entry and Add Tag buttons. I like to add three entries with these tags: try entering **hi ho** for the first entry; for the second entry, **ho hum**; and for the third entry, **ho ho ho**. Then type **ho** in the search field. The **ho** tag in the first entry is highlighted with animation for a moment, and the first instance of the **ho** tag in each entry remains selected. Press Return several times in succession. All of the **ho** tags remain selected, but the animated highlight appears on a different entry’s **ho** tag with each press of the Return key. If you spaced out the entries with lines of text, each **ho** tag in turn scrolls into view before it is highlighted.

Step 8: Build and Run the Application

Build and run the application to test once again what you've created. You can't be too careful. Even though you have run it several times while working through this recipe, it is always useful to step back and take a longer look at your work product for the entire recipe at once.

This recipe has been an exercise in using some of the features of the Cocoa text system. You have only touched the surface, but you have gotten a flavor for how to search and edit text programmatically. Use the Add Entry and Add Tag buttons to add several entries to the Chef's Diary and tag them with keywords. Type some text. To test the performance of the diary with larger amounts of text, copy a large TextEdit file and paste its entire text into each diary entry. Then play with the navigation buttons, the date picker, and the search field to see how well they perform. Be sure to undo and redo changes frequently.

Step 9: Save and Archive the Project

Quit the application, close the Xcode project window, and save if asked to do so. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 4.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 5.

Conclusion

In this recipe, you completed the Chef's Diary. More important, you learned how to add a variety of controls to a window and how to hook them up so that they actually work. This is an important part of writing any application.

In the next recipe, you will add a menu and a few menu items to the menu bar, and then hook them up to give the user alternative ways to exercise some of the controls. You will also create a Recipe Info menu item duplicating the role of the Recipe Info toolbar item in the main Vermont Recipes window. You will discover that menu item validation works almost automatically, thanks to the work you did in this recipe to validate the controls.

Documentation

Read the following documentation regarding topics covered in Recipe 4.

Class Reference and Protocol Documents

Foundation Functions Reference (NSLog)

NSDateFormatter Class Reference

NSUndoManager Class Reference

NSString Class Reference

NSControl Class Reference

NSUserInterfaceValidations Protocol Reference

NSValidatedUserInterfaceItem Protocol Reference

NSWindowDelegate Protocol Reference

NSDate Class Reference

NSDatePicker Class Reference

General Documentation

Human Interface Guidelines

Resolution Independence Guidelines

Window Programming Guide (Using Keyboard Interface Control in Windows)

Attributed Strings Programming Guide

String Programming Guide for Cocoa (Searching, Comparing, and Sorting Strings)

Data Formatting Programming Guide for Cocoa

Undo Architecture

Document-Based Applications Overview (Message Flow in the Document Architecture)

Control and Cell Programming Topics for Cocoa

User Interface Validation

Coding Guidelines for Cocoa (Code Naming Basics)

Date and Time Programming Guide for Cocoa

Configure the Main Menu

Virtually every Macintosh application outside the world of games has a main menu. It appears in the menu bar at the top of the screen when the application is active. You have already seen that it comes with the document-based application template when you create a new project, in the MainMenu nib file. It also comes with other application templates. Many of its menu items are prewired, leaving to you only the relatively simple job of hooking up those that aren't already connected and adding application-specific menu items.

In this recipe, you learn more about the first responder and the Cocoa responder chain. You also learn how to add menus and menu items to the main menu and how to hook up menu items that don't work by default. You add a completely new menu, the Diary menu, with menu items that duplicate the roles of the Add Entry and Add Tag buttons and the four navigation buttons in the window. The menu items are enabled only when the Chef's Diary window is open and active. You also add a menu item to duplicate the role of the Recipe Info button you added to the recipes window's toolbar in Recipe 2 to open the drawer, a menu item that works with the search field you added in Recipe 4, and a menu item to open a Read Me file. In the process, you learn how to make sure that menu items are properly enabled and disabled based on the changing state of the application.

In older versions of Mac OS X, you usually started fixing up the menu bar by adding the name of the application to several of the menu items supplied by the template. The About menu item in the application menu, for example, came with the title *About New Application*, and you had to replace *New Application* with the name of

HIGHLIGHTS

Creating an application controller and delegate

Using resources in the application package

Adding menus and menu items to the menu bar

More about action methods and the sender parameter

Validating (enabling and disabling) menu items

Using and manipulating the Cocoa responder chain

your application. This was also true of the Hide and Quit menu items in the application menu and the Help menu item in the Help menu. Now, however, the template fills in the application name for you.

Step 1: Create the `VRApplicationController` Class

In preparation for Step 2, create a new class, `VRApplicationController`, and designate an instance of it as the application's delegate. It is very common to create an *application delegate* class because, as you will learn later, it allows you to customize the behavior of Cocoa's `NSApplication` class without subclassing `NSApplication`. There is nothing wrong with subclassing, but Cocoa tends to favor using delegates wherever possible. `NSApplication` declares many delegate methods, and virtually all applications implement some of them in a custom application delegate class.

I prefer to call it an *application controller* because it usually serves other purposes, in addition to acting as the application's delegate. Just as the `RecipesDocument` and `DiaryDocument` classes are specialized controllers focusing on the document side of the interrelationship between the application's model and views, the `VRApplicationController` class acts as a specialized controller focusing on the application side of the interrelationship between the application at large and its windows and views. For example, you will put a menu command to open a Vermont Recipes Read Me file in the application's Help menu. Since the Help menu is available at all times, it is appropriate to put the action method that opens it in the application controller, as you will do in Step 2.

1. Leave the archived Recipe 4 project folder where it is, and open the working Vermont Recipes subfolder.

Increment the Version in the Properties pane of the Vermont Recipes target's information window from 4 to 5 so that the application's version is displayed in the About window as 2.0.0 (5).

2. Create a new class and name it `VRApplicationController`. In Xcode, choose File > New File. Select Cocoa Class in the left pane and "Objective-C class" in the right pane. Choose NSObject in the "Subclass of" pop-up menu and click Next. In the next window, enter **`VRApplicationController`** as the File Name, creating a header file to match, and click Finish to save both files in the Vermont Recipes project folder. Drag the header and implementation files from wherever they landed in the Groups & Files pane to the top of the Classes group, above the `VRDocumentController` files.

3. Open the new `VRApplicationController` header and implementation files and change the identifying information at the top according to the model you applied in Step 4 of Recipe 1. Save the header and implementation files when you're done.
4. Go to Interface Builder's Library window and select the Classes tab. There, near the bottom, you find your new `VRApplicationController` class. Drag it into the MainMenu nib file's window and drop it beside the Document Controller object you added in Step 6 of Recipe 3. As with every object in the nib file, an instance of `VRApplicationController` and an instance of `VRDocumentController` will be instantiated when the Vermont Recipes application is launched. This is appropriate, because both objects are needed as long as the application is running. You will use the new application controller in Steps 2 and 4.

Step 2: Add a Read Me Menu Item to the Help Menu

In the days of the Classic Mac OS, it was customary to include a separate read-me document in a folder holding an application and other supporting files. The read-me document explained who wrote the application, how to install it, what it does, and where to send the money. Now, in Mac OS X, applications come in the form of an application package, a folder disguised to look like a single file. A document that you formerly put in an installation folder alongside the application itself can now be put inside the application package, where it is much less likely to become separated from its owner. Some developers make an application's read-me file available outside the application by putting an alias file pointing to it alongside the application on the installation disc. That way, a new user doesn't have to launch the application to find out what it does.

To make the read-me file easily accessible to the user even while the application is running, you can add a menu item to the main menu so that the user can open the file at any time. The techniques you learn in this step can be used for other files as well, such as a quick-start document and a version-history document.

1. Create the read-me file using TextEdit or any other word processor that can save RTF text. TextEdit is installed on every Macintosh computer, so you know the user will be able to open the read-me file whether by double-clicking an alias file or choosing Help > Read Me in your application.

Open a new, empty document in TextEdit, and compose and format the content of the Vermont Recipes Read Me document. Focus on being helpful to your

users, especially first-time users. Here, we provide only a very short document intended to convey basic information and to illustrate the process (**Figure 5.1**).

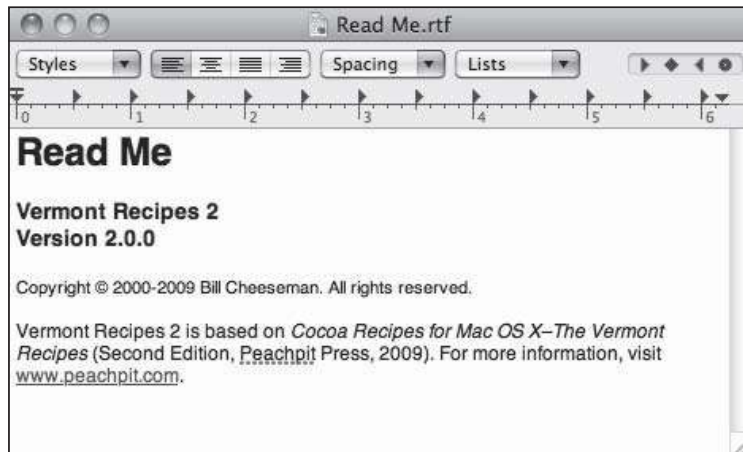


FIGURE 5.1
The Vermont Recipes
Read Me document.

Save the file in the English.lproj folder of your Vermont Recipes project folder as a Rich Text Format document, naming it *Read Me*. You could have saved it as PDF instead, since every Mac also comes equipped with Preview, which can read PDF files.

In Xcode, select the Resources group of the Groups & Files pane, choose Project > Add to Project, navigate to the English.lproj folder, select Read Me.rtf, and click Add. In the next sheet, set everything up as you have done in previous recipes and click Add. The Read Me file appears in the Resources group.

2. Write the action method. You already learned the basics in Step 6 of Recipe 3, where you wrote the `-newDiaryDocument:` action method to open a new Chef's Diary document, and you wrote several action methods in Recipe 4. All action methods have the same signature, except for the name of the method. Name this one `-showReadMe:`.

In the `VRApplicationController.h` header file, enter this declaration:

```
- (IBAction)showReadMe:(id)sender;
```

In the `VRApplicationController.m` implementation file, enter this implementation:

```
- (IBAction)showReadMe:(id)sender {
    NSString *path =
        [[NSBundle mainBundle] pathForResource:@"Read Me"
        ofType:@"rtf"];
    if ((path == nil)
        || ![[NSWorkspace sharedWorkspace] openFile:path]) {
        NSBeep();
    }
}
```

The first statement targets the application package using Cocoa's `+[NSBundle mainBundle]` class method. If you study the *NSBundle Class Reference*, you learn that Cocoa's `NSBundle` class implements a number of methods that give you access to the content of bundles, including the bundle that is the current application's package, `+mainBundle`. For example, these methods give your application the ability to use any of the resources included in the application package's Resources folder, such as images, sounds, special fonts, and, as in this case, text files. The first statement uses one of these methods, `-pathForResource:ofType:`, to get a string containing the path to the RTF file named *Read Me* and assign it to the `path` local variable. This method searches for the file in language-specific `.lproj` folders in the order specified in the user's Language & Text system preferences.

What if a file with that name is not found in the Resources folder or it has a different type? The `-pathForResource:ofType:` method returns `nil`, a common Cocoa technique for indicating that nothing was found or that some sort of error has occurred. Relying on this design pattern, the second statement tests whether the `path` variable is `nil`. If it is `nil`, the statement takes advantage of the fact that standard C employs short-circuit evaluation, exiting the expression as soon as it knows that the result is true. It skips the second test, going directly to the `NSBeep()` function. Passing `nil` into a method like `-openFile:` would cause an exception, so you should get in the habit of testing for `nil` before calling such a method, even if you don't plan to beep or do anything else with the error.

If `path` is not `nil`, the method executes the next test. This test targets a shared singleton object, `+[NSWorkspace sharedWorkspace]`, calling its `-openFile:` method with the value of `path` as its parameter. Cocoa's `NSWorkspace` class is a remarkably useful tool, providing access to the file system and the ability to open files and launch applications. Even if the file exists, some error might prevent Cocoa from opening it. In that case, `-openFile:` returns `NO`.

Finally, if both tests in the `if` clause evaluate to false, the method calls Cocoa's `NSBeep()` function and the user's computer beeps. It is important to realize that both the AppKit and Foundation implement a large number of global functions in addition to the object-oriented methods declared in Cocoa's classes. Most of these functions provide commonly used code snippets that you would otherwise have to spend time writing yourself, increasing the likelihood of errors. Some of them, like `NSBeep()`, provide access to system resources and capabilities. Part of your job when learning the Cocoa frameworks is to become familiar with these functions.

Save the header and implementation files.

3. Add a Read Me menu item to the Help menu. You added a menu item to a menu in Step 6 of Recipe 3, so you already know how to do it. First, open the `MainMenu.nib` file and click the Help menu to open it. In the Objects pane of the Library window,

choose Library > Cocoa > Application > Menus. Drag a Menu Item object into the MainMenu window and drop it immediately below the Vermont Recipes Help menu item in the open Help menu. Double-click it and change its title to *Read Me*.

The main Help menu item should stand by itself at the top of the Help menu, so add a menu item divider between the two menu items that now fill the Help menu. Drag a Separator Menu Item from the Library window and drop it between the two menu items in the Help menu.

4. Now connect the new action method and the new menu item. Control-drag from the Read Me menu item to the First Responder proxy in the MainMenu nib file's window. In the Received Actions HUD, choose the `showReadMe:` action.

You could have connected the action directly to the Application Controller icon, since the action method is implemented there. I prefer to use the First Responder proxy, whenever it works, because it gives me greater freedom to revise the application's architecture later.

5. But does the First Responder work with the Read Me menu item? To find out, save the nib file, build and run the application, and try to choose Help > Read Me. The Read Me menu item is disabled and you can't choose it.

To make the First Responder work with the `-showReadMe:` method requires one more step. Select the File's Owner proxy in the MainMenu nib file window. Recall that `NSApplication` owns this nib file. In the Application Connections inspector, drag from the `delegate` outlet to the Application Controller icon in the nib file window. You learned in Step 1 that the `VRApplicationController` class is to be the application's delegate as well as an application controller. This is how you make any object the delegate of another object using Interface Builder.

6. Save the nib file, build and run the application again, and choose Help > Read Me. This time, the Read Me menu item is enabled. When you choose it, `TextEdit` launches and your read-me file opens.

The Cocoa Responder Chain

Cocoa implements what is called a *responder chain*. Action messages sent by controls and menu items are not limited to targeting a specified object. They can optionally target the first responder. The First Responder icon in the nib file window is a proxy whose identity is determined dynamically at run time based on what the user is currently doing. Connecting a view or control to the First Responder proxy is equivalent, in code, to setting the target for the selected action message to `nil`. By deliberately failing to

(continues on next page)

The Cocoa Responder Chain *(continued)*

designate a specific object as the message's target, you tell Cocoa to find the target for you. The application itself is responsible for performing that task, using a method you saw in Step 4, `NSApplication's -targetForAction:to:from:`.

The term *responder chain* is shorthand for an algorithm that defines the hierarchy of views, controls, and other objects that Macintosh users expect to respond to actions taken in an application's user interface. Cocoa shunts every message from object to object in a chain of objects according to a well-defined path, trying to find a suitable recipient. The chain usually starts with the view or control that currently has keyboard focus in the current key window. This may be the object known as the window's *initial first responder* if the window just opened, or it may be some other view or control within the window if the user is already working in the window and clicked or tabbed to other views. For example, if the user selects some text in a text field and chooses `Edit > Copy`, Cocoa looks first at the text field and immediately sees that it can handle the `-copy:` message. The text field therefore responds.

If the focused view does not recognize the message, Cocoa does not give up but instead searches a chain of *next responders* that includes the focused view's supervIEWS, the window itself, the window's delegate, if it has one, its window controller and its document, the responder chain of the main window if another window such as a palette has keyboard focus, the application itself, and, if it has one, the application's delegate. All of the objects in the path inherit the ability to participate in the responder chain from Cocoa's `NSResponder` class, which declares the `-nextResponder` method. For example, if the user in the example above chooses `Window > Zoom` instead of `Edit > Copy`, Cocoa might see that the text field doesn't know how to do anything with the `zoom` message. In terms of code, it doesn't declare a `-performZoom:` action method. Cocoa therefore searches the responder chain from next responder to next responder until it reaches the window object. The window knows how to zoom, so it does. Similarly, if the user presses `Command-Q`, Cocoa searches the responder chain for an object that declares the `-terminate:` action method, and it finally finds it when it reaches the application object. The application quits.

The first object to recognize a message is called the *first responder*. By default, the search stops there, and the message is sent and executed. To make it possible for you to use this mechanism in Interface Builder, every nib file window includes a First Responder proxy that stands in for a target in the responder chain. This mechanism is immensely useful, because it lends simplicity and flexibility to the process of designing and implementing a dynamic and functional user interface. You, the developer, have all this power at your fingertips without having to write a lot of code.

As the sidebar explains, one of the participants in the responder chain is the application's delegate, if it has one. Although the application delegate is at the very end of the responder chain—the last place Cocoa looks for the `-showReadMe:` method—that's where Cocoa found the method once you connected the application's `delegate` outlet.

Cocoa uses the responder chain to decide whether to enable or disable any menu item that is connected to the First Responder proxy. It does this every time the user clicks a menu, just before the menu opens. If an object found in the responder chain responds to the action, the menu item is enabled; otherwise, it is disabled.

This mechanism is perfect for menu items that are only supposed to be used when a particular window is active. The responder chain always includes the active document and its active window, window controller, and window delegate.

This mechanism works for `VRApplicationController`, too. Menu items that affect the entire application should be enabled almost all the time, and `VRApplicationController` is always instantiated and therefore always in the responder chain.

Step 3: Add a Diary Menu to Control the Diary Window

It is common for Macintosh applications to duplicate in the menu bar some of the functionality of a window's controls. Arrange to do that now with the Add Entry and Add Tag buttons and the four navigation buttons in the Chef's Diary window. Create a Diary menu with the first two menu items bearing the same titles as the Add Entry and Add Tag buttons and the last four containing names derived from the `goTo` action methods of the four navigation buttons.

Proper user interface design dictates that these menu items should be disabled when the Chef's Diary window is inactive. Even when the window is closed or in the background, however, the menu itself should be enabled. This allows a user to open the menu and see the menu items, even if they are disabled, to help understand how the application works.

1. Add the Diary menu and its menu items to the menu bar.

Open the `MainMenu.nib` file, select the Objects tab in Interface Builder's Library window, and choose `Library > Cocoa > Application > Menus`. Drag a Submenu Menu Item object onto the nib file's mockup of the application's menu bar. Position it between the View and Window menu titles so that an insertion mark appears, and drop it into the menu bar.

Double-click the new menu's placeholder title, *Menu*, to select it for editing, and type **Diary**. The full name, *Chef's Diary*, would take up more space than necessary. Click the Diary menu to open it, double-click its *Item* menu item, and change its title to **Add Entry**. Drag a Menu Item object from the Library window and drop it below the Add Entry item. Rename the new menu item **Add Tag**.

Next, drag a Separator Menu Item onto the bottom of the menu, followed by another Item menu item. Next, hold down the Option key and drag the last Item menu item down until an Add (+) tag appears on the cursor, and drop the copy. Repeat this twice to make a total of four Item menu items. As you see, you can duplicate menu items by Option-dragging an existing menu item. Rename each of the four Item menu items **First Entry**, **Last Entry**, **Previous Entry**, and **Next Entry**.

2. Think about why it is appropriate to place the action methods in the `DiaryWindowController` class. The buttons are view objects in the terminology of the MVC design pattern. When clicked, the first two cause changes to be made to the MVC model, which in this case is the text in the Chef's Diary. The other four change the selection in the window. An MVC controller object is therefore the right place to write the specialized code that responds to the user's click and tells the document to update its data.

The `DiaryWindowController` is the right choice for another reason. As you just learned, it will be in the responder chain only when the Chef's Diary window is open and active. Placing the action methods in it will ensure that the new menu's menu items are enabled when the window is active, and only then.

Return to the `MainMenu.nib` file and connect the new menu items to the First Responder proxy. As you do this for each menu item, you find that its action appears in the HUD, and you are able to connect them.

3. Now you're ready to validate the new menu items, so that they are enabled and disabled at the right times. Save the `MainMenu` and `DiaryWindow.nib` files, and build and run the application.

When it's running, leave the Chef's Diary window closed and open the new Diary menu. You see that the new menu items are all disabled, as they should be because the diary window is not open. Now choose `File > New Chef's Diary` to open the diary window and make it active, and then open the Diary menu again. You see that the Add Entry menu item is enabled, and the others are disabled. Choose `Diary > Add Entry`, and a new entry title appears in the window. Open the Diary menu again, and now the Add Entry, Add Tag, First Entry, and Last Entry menu items are enabled. Choose `Diary > Add Entry` again to create a second diary entry, and then open the Diary menu again. Now the Previous Entry menu item is also enabled. Choose it, and the previous entry is selected.

Open the Diary menu again, and now the Next Entry menu item is enabled and the Previous Entry menu item is disabled (**Figure 5.2**).

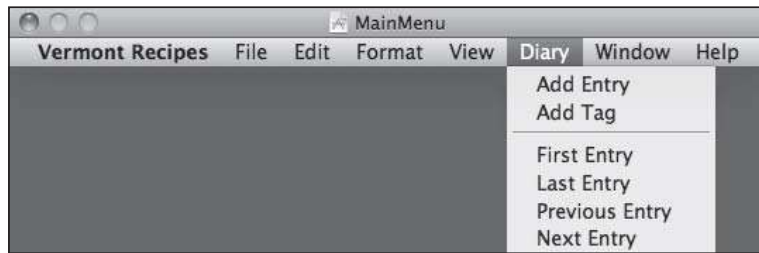


FIGURE 5.2
The Diary menu and its menu items.

In other words, you don't have to do any more work to validate the new menu items. The first responder chain takes care of disabling the menu items when the diary window is closed or inactive. When the diary window is open and active, the validation routines you wrote in Recipe 4 for the window's buttons automatically handle validation of the menu items exactly the same way, just as you anticipated when you set up the validation mechanism.

Step 4: Add a Diary Tag Search Menu Item to the Find Submenu

Applications that implement a search field in addition to a Find command commonly add a Search menu item at the top of the Find submenu in the Edit menu. Choosing the Search command does not perform the search. Instead, it makes the search field the active window's first responder. That is, it puts the blinking text insertion bar in the search field so that the user can begin typing a search term immediately.

Search menu items commonly have a Command-Option-F keyboard shortcut and are phrased as a noun rather than a verb. Look, for example, at Mail's Mailbox Search menu item and Safari's Google Search menu item. Interface Builder's Search Library menu item departs from this norm grammatically, but its behavior is similar. Their validation behavior is interesting. Interface Builder and Safari leave their Search menu items enabled when the window with a search field is closed. Their Search menu items end with an ellipsis character (...) to indicate that the command needs more information, and indeed, they do open the associated window. Mail is different. Its Search menu item does not end with an ellipsis, and it is disabled when the Mail window is closed.

In Vermont Recipes, follow the pattern of Interface Builder and Safari.

1. Start by creating the new menu item using the same technique you used in Step 3.

Open the MainMenu nib file. Using Interface Builder's Library window, drag a Menu Item object onto the nib file's mockup of the application's menu bar. Drop it at the top of the Find submenu of the Edit menu. Name it *Diary Tag Search...* (enter the ellipsis character by typing Option-semicol on a U.S. keyboard). Then drop a Separator Menu Item immediately below the Diary Tag Search menu item.

2. This menu should have a keyboard shortcut. Select the Diary Tag Search menu item. In the Menu Item Attributes inspector, click in the Key Equiv. field to select it for editing, hold down the Command and Option keys, and press the F key. The symbols for the Command-Option-F keyboard shortcut appear in the Key Equiv. field and in the menu bar mockup.

Note that you did not hold down the Shift key to type an uppercase *F*. If you had, you would have created a Command-Shift-Option-F keyboard shortcut, which is not what you want.

3. To reuse the `-findTag:` action method you wrote in Recipe 4, start by Control-dragging from the menu item to the First Responder proxy, and save the nib file. As you learned in Steps 2 and 3, choosing the menu item while the diary window is active will execute the action method.
4. You aren't done with the `-findTag:` action method, however. If you build and run the application now, and then open the diary window and choose Edit > Find > Diary Tag Search, nothing happens.

It's a good idea to keep the debugger console window open as a debugging aid in situations like this. In Xcode, choose Run > Console and position the console window where you can see it. When you choose the Diary Tag Search menu item again, you see a long error message in the console window telling you that an unrecognized selector, `-[NSMenu stringValue]`, was sent. Looking at the `-findTag:` action method, you see that one of its first statements obtains the tag string to search for by calling `[sender stringValue]`.

It is all right to send a `-stringValue` message to the `sender` when the `sender` is a search field control. `NSSearchField` responds to the `-stringValue` message, returning the `NSString` object representing the search term entered by the user. Now, however, that the `sender` is an `NSMenuItem`, which does not respond to `-stringValue`.

The problem is that the new Diary Tag Search menu item is not intended to perform a search, which is what you designed `-findTag:` to do. Instead, the menu is supposed to make the search field the diary window's first responder, opening the diary window if it isn't already open, so that the user can type a search term.

In the `DiaryWindowController.m` implementation file, revise the `-findTag:` method by adding these three lines to its beginning:

```
if ([sender isKindOfClass:[NSMenuItem class]]) {  
    [[self window] makeFirstResponder:[self searchField]];  
} else {
```

Close the new `else` clause by inserting a closing brace (`}`) at the end of the method.

You could have tested whether the sender responds to the `-stringValue` method, like this:

```
if ([sender respondsToSelector:@selector(stringValue)]) {
```

However, this will be rather opaque when you come back to look at your code a month from now. You are adding this new branch to the `-findTag:` method specifically to provide alternate behavior for the new menu item, and your purpose is clearer if you test specifically for the `NSMenuItem` class.

`NSWindow`'s `-makeFirstResponder:` method is the standard Cocoa technique to move keyboard focus to a different view. Sometimes you pass `nil` to the method in order to make the window itself the first responder. This has the effect of *committing* the value of the old first responder and triggering a number of delegate methods. Here, however, you want to give keyboard focus to a specific control, the search field.

5. You must complete one last task to make the `-findTag:` method work as the Diary Tag Search menu item's action method. You must implement an outlet and accessor method for the search field; otherwise, the method won't compile because `self` (the window controller) has no `-searchField` method.

You've created outlets before. In the `DiaryWindowController.h` header file, add this outlet declaration between the braces of the `@interface` directive, after the existing `datePicker` outlet:

```
IBOutlet NSSearchField *searchField;
```

Declare its accessor method below the `-datePicker` accessor:

```
- (NSSearchField *)searchField;
```

Implement it in the `DiaryWindowController.m` implementation file like this:

```
- (NSSearchField *)searchField {  
    return [[searchField retain] autorelease];  
}
```

In Interface Builder, in the `DiaryWindow.nib` file, connect the new outlet by Control-dragging from the File's Owner proxy to the search field and choosing `searchField` from the Outlets section of the HUD.

6. The new Diary Tag Search menu item now works. Also, as you learned in Step 3, it is already validated. Build and run the application now, open the Chef's Diary window, and try to choose `Edit > Find > Diary Tag Search`. You can't choose it, because it is disabled when the diary window has no tag lists. You arranged for this behavior in Step 7 of Recipe 4, when you added an `else if` clause to the `-validateUserInterfaceItem:` protocol method to test whether the `-firstTagRange` method found a tag list. Click Add Entry and Add Tag to add a new entry with a tag list. Now choose `Edit > Find > Diary Tag Search` again. The menu item is now enabled, and when you choose it, the search field acquires keyboard focus, and you can immediately begin typing a search term.
7. In Step 3, you were done implementing the Add Entry, Add Tag, and navigation menu items when you reached this point. Here, however, you have not yet created the behavior you set out to implement for the Diary Tag Search menu item. To see why, close the diary window in the running Vermont Recipes application, and choose `Edit > Find > Diary Tag Search`. You can't, because the menu item is disabled. You want it to be enabled all the time, so the user can choose it to open the diary window when it is closed and immediately search for a tag.

The problem and its solution are very simple, and they show off the importance of the Cocoa responder chain. As you learned in Step 1, a menu item that is connected to the First Responder proxy (that is, a menu item whose target is `nil`) is enabled only if its action method is found in the current responder chain. The search field's action method, `-findTag:`, is implemented in the `DiaryWindowController` class, which is not in the responder chain when the diary window is closed. The Diary Tag Search menu item is therefore disabled.

The solution is to implement it in the `VRApplicationController` class, too. The existence of two action methods with the same name does not create a conflict. Instead, the responder chain is designed to deal with exactly this situation. When the diary window is open, the `DiaryWindowController` is in the responder chain. When the user chooses `Edit > Find > Diary Tag Search`, the

menu item is enabled because the `-findTag:` action method in `DiaryWindowController` is found in the responder chain, and it is executed. It doesn't matter if there is another `-findTag:` action method farther along in the responder chain, in the application controller, because Cocoa stops looking for action methods when it finds the one in the window controller. When the diary window is closed, the same search of the responder chain goes all the way to the application controller before it finds a `-findTag:` action method. That is enough to enable the Diary Tag Search menu item, even when the diary window is closed, and choosing the menu item executes the version of the `-findTag:` action method implemented in the application controller.

Before implementing the action method in the application controller, resolve a minor issue in the `-validateUserInterfaceItem:` method. As it exists now, it disables the Diary Tag Search menu item and the search field control if there are no tags in the diary's text. This is inconsistent with the notion that you can institute a tag search even when the diary window is closed, because the diary window that the menu item opens might be empty. Choosing Diary Tag Search would open the diary window, but it would not give keyboard focus to the search field because the search field is disabled when there are no tag lists. This will be disconcerting to the user, so make the search field available even when there are no tag lists in the diary. The similar search menu item in Safari works the same way; it is enabled even if no Web site is showing in the window. This is not illogical; the search for a tag will come up empty, which is what always happens when you search for a search term that is not found.

To eliminate validation of the Diary Tag Search menu item and the search field while the diary window is open, simply remove the last `else if` clause from the `-validateUserInterfaceItem` method.

8. Now you're ready to implement a second `-findTag:` action method, this time in the `VRApplicationController` class. Declare it in the `VRApplicationController.h` header file, just after the existing `-showReadMe:` action method, like this:

```
- (IBAction)findTag:(id)sender;
```

Implement it like this in the `VRApplicationController.m` implementation file:

```
- (IBAction)findTag:(id)sender {
    [[VRDocumentController sharedDocumentController]
        newDiaryDocument:sender];
    NSWindow *keyWindow = [NSApp keyWindow];
    if ([[keyWindow windowController] isKindOfClass:
        [DiaryWindowController class]]) {
```



```

        DiaryWindowController *keyWindowController =
            [keyWindow windowController];
        [keyWindow makeFirstResponder:[keyWindowController searchField]];
    } else {
        NSBeep();
    }
}

```

The first statement sends the `-newDiaryDocument:` message to the singleton `sharedDocumentController` object that every document-based application implements. You wrote the `-newDiaryDocument:` action method in Step 5 of Recipe 3. Here, you pass the `sender` of the `-findTag:` action method along to the `-newDiaryDocument:` action method, as you should always do when you call one action method from another. This statement causes the diary window to open, or to come to the front if it is already open.

The rest of the statements test whether the diary window really opened and became active. If it did, the method makes the search field the first responder. To test whether the diary window is open and active, you get the application's key window, which by definition is the window currently having keyboard focus, and make sure its controller is `DiaryWindowController` using the `-isKindOfClass:` method you've used before. If so, you use the `-makeFirstResponder:` method you first encountered a moment ago to make the search field the first responder of the window. If the `-findTag:` method somehow failed to open the diary window, the method causes the computer to beep, a common though uninformative signal to the user that something went wrong.

The statement that calls `-makeFirstResponder:` could just as well have been written to call `DiaryWindowController`'s version of the `-findTag:` action method, passing `sender` along. Since that version of `-findTag:` would see that `sender` is an `NSMenuItem` object, it would execute the first branch of the `if` clause that you just wrote, which itself calls `-makeFirstResponder:`. This is why it is important to forward the `sender` parameter value whenever you call one action method from another. Calling the other `-findTag:` action method has two advantages: It forces the behavior of the two action methods to remain identical with respect to setting the first responder, and it reminds the reader that the application controller's action method has a counterpart in the diary window controller. Keeping the behavior of the two classes synchronized is important. If you ever change the window controller version of the action method, the application controller will acquire the same new behavior automatically. You should therefore go ahead and substitute this statement now:

```

[keyWindowController findTag:sender];

```

9. If you try to build the project now, Xcode will complain that it never heard of the `VRDocumentController` and `DiaryWindowController` classes. To cure that problem, add these two lines after the first `#import` statement at the top of the `VRApplicationController.m` implementation file:

```
#import "VRDocumentController.h"
#import "DiaryWindowController.h"
```

The Diary Tag Search menu item now works as specified at the beginning of this step. It is enabled and disabled at all the right times, and it makes the search field the first responder whether you call it with the diary window open or closed.

Step 5: Add a Recipe Info Menu Item to Open the Recipes Window's Drawer

In this step, you add a Recipe Info command to the Window menu to complement the Recipe Info button you added to the recipes window's toolbar in Step 6 of Recipe 2. When the recipes window is active, the user should be able to use the Recipe Info menu item to open and close the drawer instead of using the toolbar. The user might prefer to keep the toolbar closed to make more room, so duplicating the Recipe Info button's functionality in the menu bar makes sense.

1. To start, open the `MainMenu` nib file, and in the mockup of the menu bar, open the Window menu. It already contains three menu items from the template: Minimize, Zoom, and Bring All to Front. When the application is running, it also has menu items at the bottom, added automatically by Cocoa on the fly as you open application windows.
2. Select the Objects tab in the Library window and choose Library > Cocoa > Application > Menus. Drag a Menu Item object to the Window menu, drop it below the Zoom menu item, and retitle it *Recipe Info*. Also drag a Separator Menu Item from the window and drop it between Zoom and Recipe Info.
3. Connect the new Recipe Info menu item to its action. You already know from Step 6 of Recipe 3 that `NSDrawer's -toggle:` action method is the one you want. Control-drag from the Recipe Info menu item to the First Responder in the `MainMenu` nib document window. In the HUD, scroll down to the toggle: action and select it.

Unfortunately, if you build and run the application now, you discover that the Recipe Info menu item remains disabled even when the recipes window is active. Your work to date suggests that this is a responder chain issue. The `-toggle:`

method is implemented in the `NSDrawer` class, but nothing in the documentation indicates that a drawer is part of the responder chain by default. When you open the Window menu and Cocoa searches the responder chain, it does not find the `-toggle:` method and therefore disables the Recipe Info menu item.

The Recipe Info button in the window's toolbar worked only because you didn't rely on the responder chain. Instead, you connected the button directly to the drawer where the `-toggle:` method is implemented. This worked because the button and the drawer were in the same nib file. Now, for the menu item, you have to start with the `MainMenu` nib file where the menu bar resides, and you can't drag to the separate `RecipesWindow` nib file to connect the menu item to the drawer.

There are several commonly used solutions to this issue in the general case. You know from your work in Step 3 that you should focus on the `RecipesWindowController` class, because the Recipe Info menu item should be enabled only when the recipes window is active. Consider several possibilities first; then settle on the most elegant solution.

Put everything in one nib file. In a sense, what you've just encountered is a fundamental restriction in Interface Builder. You can't connect objects in separate nib files.

One solution, therefore, is to place the `RecipesWindowController` object and related objects such as the window and the drawer in the same nib file. In fact, applications are often written with the menu bar and the application's main window in a single nib file, along with auxiliary objects such as a drawer. In those applications, you would simply connect the menu item to the drawer and select the `-toggle:` method, ignoring the responder chain, and you would be done.

Set the target and action programmatically. You can do in your own code what Interface Builder does for you automatically. Cocoa's `NSMenuItem` and `NSControl` classes and several other classes implement `-setTarget:` and `-setAction:` methods. If you implement outlets identifying the players and set up the classes in your project so that they know how to talk to one another, you can make the drawer the target of the menu item and make the action method's selector its action. By doing this, you avoid the responder chain completely.

Write an intermediary action method. You can write a custom action method in a class that is in the responder chain by default, and have it call the drawer's `-toggle:` method. This lets you take advantage of the flexibility and power of the responder chain and the ease of use of Interface Builder to connect the menu item to the First Responder proxy in the `MainMenu` nib file.

The only code you have to write to do this is the custom action method itself. You would put it in `RecipesWindowController`, which you already know from Step 3 is in the responder chain by default. You might call it `-toggleInfoDrawer:`, and in its implementation simply call the drawer's built-in `-toggle:` method.

This is in fact a commonly used technique. It has the disadvantage, however, of requiring you to write an intermediary method like `-toggleInfoDrawer:` every time you want to call a method that the Cocoa engineers have already written for you. There is a more elegant solution, which you implement now.

Add the drawer to the responder chain. The responder chain has a default configuration, which you have taken advantage of several times. Objective-C and Cocoa are very dynamic, however, and it should not surprise you to learn that you can alter the responder chain's configuration at will.

Here, a clue to the most elegant solution is the fact that `NSDrawer` is a subclass of `NSResponder`. In other words, `NSDrawer` is designed to be part of the responder chain. However, Apple does not put drawers in the responder chain by default, most likely because Apple can't know in advance exactly how you will use it. You will learn later that another important class, `NSViewController`, is also a subclass of `NSResponder` but also is not part of the responder chain by default. You are free to add `NSDrawer` or `NSViewController` to the responder chain in your applications whenever it fits your design goals.

Once your application inserts the recipes window's drawer into the responder chain, you can call the drawer's action methods just as you call any action method in the responder chain. After you connect the Recipes Info menu item to the First Responder proxy in the MainMenu nib file, Cocoa sends the menu item's action, `toggle:`, to its target, the drawer, by using the responder chain mechanism. You don't have to write an intermediary action method or an outlet in `RecipesWindowController`. By writing code once to add the drawer to the responder chain, you save yourself the trouble of writing an intermediary action method every time you want to connect another action to an object in the drawer. When you add views and UI elements to the drawer later, any action methods they implement will work simply by connecting them to the First Responder proxy.

The key to making this work is Cocoa's `-[NSResponder setNextResponder:]` method. Be careful about calling it by itself, however. If you use this method to insert an object into the responder chain and stop there, you may break the chain and effectively disconnect objects beyond the point where you inserted the new object. Rather than go to the trouble of testing whether a particular object has a next responder that must be reconnected, simply get in the habit of always saving the next responder temporarily, then reconnecting it immediately after splicing in the new next responder. If the next responder was `nil` to start with, patching `nil` onto the new next responder is harmless. But if it was not `nil`, you've just saved yourself a lot of debugging time.

The traditional place to write the code that alters the responder chain is in your implementation of Cocoa's `-awakeFromNib` method. When an application loads a nib file, it first initializes the nib file's owner and takes care of some related housekeeping. During this process, there is no guarantee that all of the objects in the nib file have been instantiated or that all of their connections have been made, so you should not place methods that rely on the nib file's objects and connections in the file's owner's initialization methods. As soon as all of the objects and their connections are ready, Cocoa calls the file's owner's `-awakeFromNib` method. You can safely override it in your subclass and place a call to the inherited `-setNextResponder:` method there. In the case of window controllers, however, Apple engineers informally encourage developers to override `-[NSWindowController windowDidLoad]`, instead of `-awakeFromNib`, because `-awakeFromNib` may be called multiple times. You learned this when you implemented an override of `-windowDidLoad` in `DiaryWindowController` in Step 7 of Recipe 3.

This time, override the `-windowDidLoad` method in `RecipesWindowController`. You don't have to declare it, because `NSWindowController` has declared it in its header file already. Add this method implementation to the `RecipesWindowController.m` implementation file:

```
- (void)windowDidLoad {
    NSResponder *oldNextResponder = [self nextResponder];
    NSResponder *newNextResponder =
        [[[self window] drawers] objectAtIndex:0];
    [self setNextResponder:newNextResponder];
    [newNextResponder setNextResponder:oldNextResponder];
}
```

I have named the local variables `newNextResponder` and `oldNextResponder` to emphasize their roles in the splice operation and to generalize the code. If you splice another object into the responder chain in another class, you will be able to copy the body of this method verbatim, changing only the part of the code in the first statement that obtains the object to be inserted.

The first statement temporarily saves the window controller's current next responder in the `oldNextResponder` local variable. You will patch it onto the drawer, the `newNextResponder`, in the fourth statement. The `oldNextResponder` might be anything, as long as it is a subclass of `NSResponder` and thus eligible to participate in the responder chain. It might even be `nil`.

The second statement saves the drawer in the local variable `newNextResponder`, after getting it from the array returned by the window's `-drawers` method.

The local variable is typed as `NSResponder*` to make the code transportable to another method without forcing you to change the object type. You could have typed it here as `NSDrawer*`.

The third statement replaces the window controller's next responder with the drawer, `newNextResponder`, splicing it into the responder chain.

The fourth statement patches the `oldNextResponder`, whatever it might be, onto the drawer, `newNextResponder`, which is now the window controller's next responder. This restores the integrity of the responder chain.

4. Once again, you don't have to do anything to ensure that the new Recipe Info menu item is properly validated. It is disabled when the recipes window is closed or not active, because under those circumstances `RecipesWindowController` and the drawer are not in the responder chain.
5. Build and run the application to test it. With the recipes window active, choose `Window > Recipe Info`, and the drawer opens. Choose it again, and the drawer closes. Close the recipes window or open the Chef's Diary window in front of it, and the Recipe Info menu item is disabled.

Step 6: Build and Run the Application

In this recipe, you built and ran the application at the end of almost every step to make sure things were working. It remains important to build and run it at the end of the recipe, to confirm and review what you've done.

Choose `Help > Read Me`, and the Read Me file opens in `TextEdit`. This works even if none of Vermont Recipes' windows are open. While making this work, you learned that you can design a class to act both as an application controller and as a delegate of the application. Making it the application's delegate had the side effect of putting it in the application's responder chain. As a result, you were able to connect a menu item having application-wide scope to an action method in the application controller simply by connecting the menu item to the First Responder in the `MainMenu.nib` file. This will come in handy every time you need to add an application-wide menu item to the menu bar.

Open the Chef's Diary window and choose `Diary > Add Entry`. The menu item is enabled when the window is active, and it adds a new entry when you choose it. While making this work, you learned that any menu item that should only be available while a related window is open should be connected to an action method in the

window's window controller. The window controller is in the responder chain only while its window is active. When the window is not active, the menu item is disabled because Cocoa does not find its action method in the responder chain. Again, all you have to do to make the menu item work is to connect it to the First Responder proxy in the MainMenu nib file. The Add Tag menu item and the navigation menu items work similarly.

The Diary Tag Search menu item works both when the diary window is open and when it is closed, because you implemented its action method in the diary window's controller as well as in the application's controller. The application controller is always in the responder chain, so the action method is always enabled. Which version of the action method is executed depends on whether the diary window is open or closed, which determines what version of the method is encountered first in the responder chain.

Finally, open the recipes window and choose Window > Recipe Info. The window's drawer opens, and it closes when you choose the menu item again. While making this work, you learned that some user interface objects, such as drawers, are not in the responder chain by default. To make it possible to call their action methods simply by connecting the menu item to the First Responder proxy in the MainMenu nib file, you may splice the interface object into the responder chain using a more general technique than you used with the application controller.

Step 7: Save and Archive the Project

Quit the running application, close the Xcode project window, and save if asked. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 5.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 6.

Conclusion

You have learned several valuable techniques to leverage the responder chain to make the application's menu bar work. In the next recipe, you will first organize the project's source files by inserting markers that make it easier to find your way around. Then you will refine the behavior of the Chef's Diary.

Documentation

Read the following documentation regarding topics covered in Recipe 5.

Class Reference and Protocol Documents

NSBundle Class Reference

NSWorkspace Class Reference

Foundation Functions Reference (NSBeep)

NSResponder Class Reference

General Documentation

Application Menu and Pop-up List Programming Topics for Cocoa

Cocoa Event-Handling Guide

Action Messages

Control the Document's Behavior

To get the application to its current state, you focused on the important functions of the Chef's Diary. The diary document can be edited, saved, and reopened; the text of the diary can be edited in either pane of the split view; the controls in the diary's window work as expected to add entries and tags, to navigate between entries, to select entries by date, and to search for entries by tag; and the menu bar includes a number of new features that support the diary window.

There remains one important issue, however. Have you noticed that the user can create multiple new diary documents? This is common in most applications, of course, and that is why document-based applications exhibit this behavior by default. But it is contrary to the general understanding of a diary. A diary should be a single document to which the chronicler adds information as time goes by. In this step, you deal with this problem by refining the application specification to require that the user record all gastronomic experiences in a single diary document.

You start by organizing your code to make it more understandable. Then you take steps to make sure the user can open one, and only one, Chef's Diary document, no matter where or under what name the user initially saved it. For flexibility, you nevertheless allow the user to switch to another Chef's Diary document when desired—for example, to allow the user to create and use backups. In the process, you learn how to deal gracefully with document-handling errors.

HIGHLIGHTS

Using Objective-C categories to organize code

Using #pragma mark statements to organize code

Creating and resolving alias records

Creating and resolving bookmarks in Snow Leopard

Saving values in user defaults and retrieving them

Setting up a one-of-a-kind document

Handling errors in a document-based application

Preparing localizable strings for internationalization

Step 1: Organize the Project's Code

Before adding all the additional methods you will implement in this recipe, it would be a good idea to organize its code. It has gotten moderately large already, and it is in danger of becoming unwieldy or even unreadable if you don't impose some organizing principle on it.

So far, the only organizing principle the book has explained is the larger structure of the classes themselves. The Cocoa frameworks are organized into two major groups, Foundation and the AppKit, having distinct functionality. As they are described in the *Cocoa Fundamentals Guide*, Foundation classes focus on data and operations on data that are unrelated to the graphical user interface, while AppKit classes focus on views and functionality that relate to the graphical user interface. The names and inheritance relationships of these classes make it easy to categorize the methods they implement according to the nature of the work they do. The classes of the Vermont Recipes application participate in this organizing principle by conforming to the structure imposed by the Cocoa frameworks, mostly because they are subclasses of built-in Cocoa classes.

It is important to impose structure on the methods within any one class, as well, especially as individual classes get larger. Cocoa and Objective-C provide three principal techniques for imposing order on any class. One is the Objective-C language feature known as the *category*. Another is the *pragma mark*, which makes use of the C language feature known as a *pragma*. The third is the lowly *comment*, which does not require much discussion. Cocoa developers frequently use all three techniques. They allow you to group similar or related methods together and to give a descriptive name to each of the groups.

Categories

Start with categories. A category adds methods to a specific class that is declared separately. A category is set off with its own `@interface` directive naming the class, just like the class's `@interface` directive, but it adds the category's name in parentheses. The methods within a category of a class work exactly as they would if they were declared and implemented in the main body of the class. One significant use of a category is to help organize the methods of a large class into more manageable and understandable segments, each with a category name describing its purpose or function.

What purpose does this mechanism serve that wouldn't be equally well served by simple comments interspersed in the code? One is that it is easy and convenient to place a category declaration and its methods in a separate file. You have to use an `#import` control line to inform the build system where to find the class declaration,

but the category's `@interface` directive and name provide an easily understood place to break a potentially large file into small pieces and give them useful names.

A related but sometimes more important purpose is that categories allow you to declare some of a class's methods in the implementation file, along with the class's `@implementation` directive and its method definitions. This is particularly useful in frameworks, because developers generally distribute a framework's header file but do not distribute the implementation file. This technique allows the developer to keep internal methods more or less secret from the frameworks' clients. Clients should not rely on private methods declared in this manner, leaving the developer of the framework free to create new versions that use different private mechanisms. Unless you work for Apple, you don't have access to the `NSApplication` implementation file, but it is a good bet that the implementation file contains one or more `@interface` directives in this form:

```
@interface NSApplication(PrivateMethods)
```

Another purpose of categories is to make it easier for a developer to see the structure of a class at a glance. Open the function menu in the `NSApplication.h` editor window, and you see all of the category declarations in boldface. This makes it much easier to scan the menu for methods that are grouped in any one of the categories.

`NSApplication` declares a total of eight categories: `NSWindowsMenu`, `NSFullKeyboardAccess`, `NSServicesMenu`, `NSServicesRequests`, `NSServicesHandling`, `NSStandardAboutPanel`, `NSApplicationLayoutDirection`, and `NSDeprecated`.

One of these categories, `NSServicesRequests`, is a category on another class—namely, `NSObject`. It is quite common to declare categories on `NSObject` in another class's header file. `NSApplication` declares three categories relating to the standard Services menu, but one of them contains methods that are useful in connection with many classes. Because they are declared in `NSApplication` as a category on `NSObject`, they are part of the repertoire of every class that inherits from `NSObject`. At the same time, placing the declaration of the category in the `NSApplication` header file makes clear that these methods relate to functionality—the Services menu—that is fundamentally related to `NSApplication`.

Currently, none of the classes you have written for Vermont Recipes declares a category. Most of the application's classes inherit from existing Cocoa classes, and most of the application's methods fit into the existing structure of the Cocoa classes and categories. Eventually, when you implement AppleScript support, you will declare some AppleScript-related categories as a convenient, though not necessary, technique to segregate AppleScript support methods from the other methods in the Vermont Recipes header files.

Categories

Categories are a feature of the Objective-C language not commonly found in other programming languages. Categories can be used in a variety of ways.

In their most interesting use, categories allow you to extend and enhance the functionality of any class, even if you don't have access to its source code, by adding new methods to the class or reimplementing existing methods it already implements. When what you want to do is add a few methods to an existing class, categories may be a good substitute for subclassing it even if you do have access to its source code. New methods implemented in a category become available to all subclasses of the base class, and they are indistinguishable at run time from methods implemented in the base class. You can add both class methods and instance methods in a category. However, you cannot add new instance variables, and when you redefine an existing method in a category, you cannot access the original method as you could by sending a message to `super` if you had subclassed the base class.

Categories on one class are often declared and implemented within header and implementation files for other classes, usually because the methods in the category relate to the function of the class in whose files they appear. You can see many examples of this in the Cocoa frameworks by browsing their header files.

Multiple categories on one class can also be declared and implemented in the same file as the base class, as a device to break the class into convenient topical sections. You can use categories in this way to partition the implementation of a single class into separate implementation files. You import the common class header in each of the implementation files. This is particularly convenient for managing a large, complex class.

Another use of categories is to declare *informal protocols*, a subject you will learn about later.

Finally, you can declare and implement a category in separate files of its own. For example, you might create a category on `NSString` in separate header and implementation files, which could serve as a reusable string library with custom methods for your private use.

Objective-C 2.0 allows use of a new device similar to a category known as a *class extension*. You declare a class extension the same way you declare a category, except that you leave the parentheses after the class name empty. The implementations of methods declared in a class extension must appear in the main `@implementation` block of the class. Class extensions, like categories, enable you to declare additional methods for a class separately from the main `@interface` block, but they provide increased error detection at compile time.

The `#pragma mark` Statement

A simpler technique for organizing and labeling sections of code within a class is the *#pragma mark statement*. Pragas are recognized by the C preprocessor as instructions to perform an implementation-dependent action. The pragma of interest here is the `#pragma mark` statement. It is in part a simple mechanism to insert custom headings into your code. More important, the `#pragma mark` statement serves to organize the function menu in the Xcode editor window's navigation bar by inserting markers in the menu.

The function menu automatically includes the name of every class, protocol, category, extension, function, and method declared or implemented in a code file, as well as `#define` directives and type declarations. It is ordered either alphabetically or by the order in which the names appear in the code file, depending on an Xcode preference setting. You can choose the other setting temporarily by holding down the Option key while you open the menu. The function menu is a very convenient way to navigate quickly to a particular method or other code element in a code file. However, like the code file itself, the menu can become very long and unwieldy.

The `#pragma mark` statements discussed here not only act as section headings in your code but also cause the text association with them to appear as markers in the function menu in order to break the menu items into understandable groups.

In this section of Step 1, you insert useful `#pragma mark` statements in both the interface and implementation parts of almost all of the classes you have declared. The organization of methods in the classes and the text and style of the `#pragma mark` statements are those that I find convenient in my work. You are free to use whatever organizing principles and naming conventions you find useful.

All `#pragma mark` statements take effect as soon as you enter them. You don't have to save or build the project. You can therefore see each of the markers by opening the function menu as soon as you have entered them.

1. Before beginning, update the project's version, as you do in each recipe. Leave the archived Recipe 5 project folder where it is, and open the working Vermont Recipes subfolder.

Increment the Version in the Properties pane of the Vermont Recipes target's information window from 5 to 6 so that the application's version is displayed in the About window as 2.0.0 (6).

2. Start by opening the `DiaryWindowController.m` implementation file, which implements more methods than almost any of the other classes you have written. Insert `#pragma mark` statements at the locations described below.

Before the `-init` method:

```
#pragma mark INITIALIZATION
```

Before the `-diaryView` method:

```
#pragma mark ACCESSOR METHODS
```

Before the `-addEntry:` method:

```
#pragma mark ACTION METHODS
```

Before the `-windowDidLoad` method:

```
#pragma mark OVERRIDE METHODS
```

Before the `-textDidChange:` method:

```
#pragma mark DELEGATE METHODS
```

Before the `-updateWindow` method:

```
#pragma mark USER INTERFACE VALIDATION
```

Before the `-keyDiaryView` method:

```
#pragma mark UTILITY METHODS
```

Before the `-currentEntryTagRange` method:

```
#pragma mark DIARY ENTRY TAG RANGE METHODS
```

Before each of the three `@implementation` directives for the validated control subclasses:

```
#pragma mark -
```

Open the function menu (**Figure 6.1**), and you see that a marker consisting of a single hyphen is displayed as a menu divider.

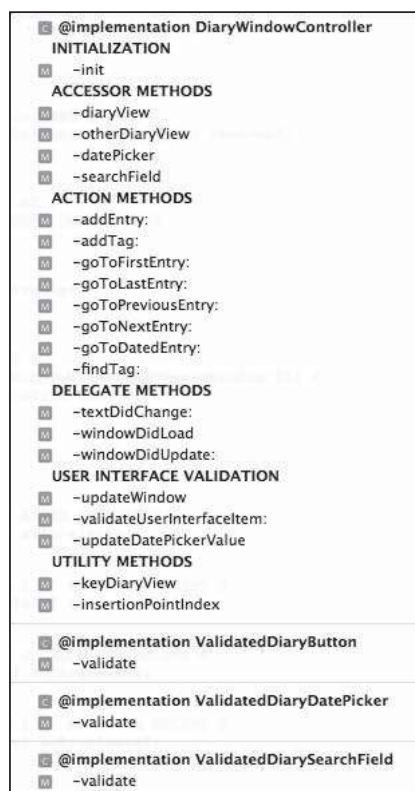


FIGURE 6.1. The function menu showing markers from the `DiaryWindowController.m` implementation file.

3. Insert identical markers at the corresponding locations in the `DiaryWindowController.h` header file. There is no need for markers for the macros, initialization, and delegate method groups because they don't appear in the header file.

Insert `#pragma mark` statements for menu item dividers before the two protocols that are declared in the header. They have no corresponding presence in the implementation file.

4. Insert `#pragma mark` statements in all of the other header and implementation files according to the same plan. I won't spell them out here, but you can see them in the downloadable project file for Recipe 6.

In addition to `#pragma mark` statements, certain words that you place in comments in your code files cause those words and the remainder of the comment to appear in the function menu. Because the remainder of the comment appears in the menu, it is a good idea to use short comments, similar to the typically short markers used with `#pragma mark` statements. These words must appear exactly as shown here, including capitalization and the trailing colon: `MARK:`, `TODO:`, `FIXME:`, `!!!:`, and `???:`. **Figure 6.2** shows how comments like these appear in the function menu, based on code you will write in Step 2.

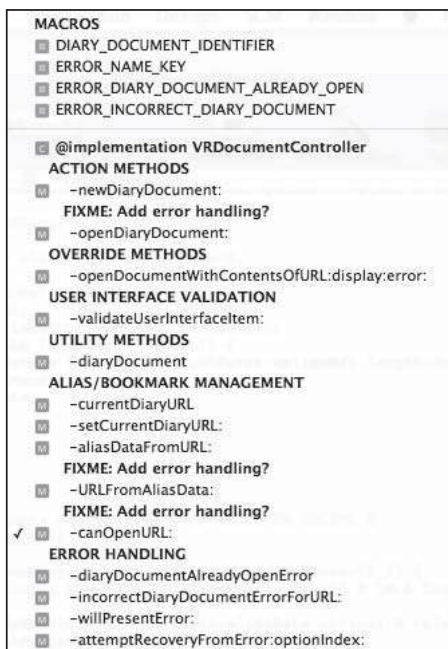


FIGURE 6.2. The function menu showing comments from the `VRDocumentController.m` implementation file after adding methods in Steps 2 and 3.

Step 2: Limit the Application to a Single Diary Document

Now that you have organized the Vermont Recipes code, turn back to the Chef's Diary document. Although the document is fully functional, you need to adjust its behavior so that the user always works with a single diary document.

To implement this feature, you apply the concept of the *current* diary document. The user has complete freedom at the beginning to give the document any name

and to save it in any location, and the user remains free to rename and move it later. However, that document, under whatever name and wherever located, remains the one and only Vermont Recipes Chef's Diary until the user explicitly says otherwise.

As a first requirement, the application must at least prevent the user from opening multiple diary documents at once. But more is required. Users can always move, rename, or duplicate files in the Finder, and the application's own Save As menu item allows the user to create duplicates. Arrange for the application to record in its user defaults or preferences the name that the user first chooses for the diary document and the location where the user first saves it. The name and location are saved in an alias record or bookmark so that the application can track the document if it is renamed or moved. In this way, the user remains able to open the document at any time simply by choosing a menu item, without having to remember the document's name or location or to go looking for it. In case the user creates a duplicate diary document—say, for backup—the application prevents the user from opening any diary document other than the one recorded in user defaults as the current diary document, unless the user provides confirmation of intent to switch to a different diary document.

Consider the specific user interface features needed to implement this concept of the current Chef's Diary. A user who has never before created a Chef's Diary must be able to use the New Chef's Diary menu item to create one. This means that the menu item must be enabled by default, as it already is. If a Chef's Diary document has already been created and is currently open, the New Chef's Diary menu item should be disabled to prevent the user from creating another. Once the user has saved the document, the name of the New Chef's Diary menu item should change to Open Chef's Diary. Thereafter, the menu item should remain enabled at all times, to open the document if it is currently closed or to bring the document's window to the front if it is already open.

Other applications that specify a single document often require the user to save it using a fixed name and a fixed location. The name is often obscure, such as *domain* in Apple's iWeb application. Its location is often one that is not readily accessible to most users, such as the Application Support folder, so that the user isn't tempted to rename it, move it, or create multiple copies of it. That approach is not foolproof, however, and it seems unduly restrictive. For example, it can cause difficulty for users who forget to copy documents in these odd locations when they upgrade to a new computer or restore from a backup.

Instead, the user should be allowed to save the Chef's Diary anywhere and under any name, as is the case with most document-based applications. At the same time, since there can be only one current Chef's Diary document, the user should be allowed to reopen it without having to remember its name or location.

Vermont Recipes accomplishes this by remembering where and under what name the document was initially saved. To cover all the possibilities of independent action by the user, the application tracks the file and changes the remembered information if the user moves or renames the document using the Finder, AppleScript, or any other technique, or if the user creates a copy of the file in a new location or under a new name using the Save As menu item. Finally, the application asks for confirmation if the user attempts to open a different copy of the document, by using either the application's Open menu item, the Finder, or some other application.

Persistent application state information such as the name and location of a file is usually saved using Cocoa's user defaults mechanism, and that is what Vermont Recipes does. In order to take advantage of the built-in ability of the Mac OS X file system to track files even if they are moved or renamed, you save an alias record or, in Snow Leopard, a bookmark in the application's user defaults. For now, the only user interface you provide to let the user choose a different document as the current Chef's Diary is the Save As menu item. Eventually you will implement a more specific user interface in the application's preferences window.

The functionality you develop in this recipe may work as well if you were to use `NSDocument`'s built-in `-fileURL` and `-setFileURL:` methods, introduced in Mac OS X 10.4, instead of the `-currentDiaryURL` and `-setCurrentDiaryURL:` methods used here. However, the `NSDocument` methods are not documented sufficiently to be sure they can be relied on for everything you do here. In addition, implementing the techniques shown in this recipe is an opportunity to learn several important lessons, including how document-based applications work, how to use alias records and Snow Leopard bookmarks, how to read and write information in the Cocoa user defaults object, and how to implement error handling for documents.

1. Start by disabling the New Chef's Diary menu item if a diary document is currently open and has not yet been saved. This is the only situation in which the menu item will be disabled. Once the document is saved, the menu item's title changes to *Open Chef's Diary*, and its action changes to a new `-openDiaryDocument:` action method. The Open Chef's Diary menu item should remain enabled at all times, either to open the document if it is closed or to bring it to the front if it is already open.

In Step 4 of Recipe 4, you learned how to validate a control using two custom protocols that you created, `ValidatedControl` and `ControlValidations`. It was necessary to implement custom protocols to validate the controls in the diary window because `NSControl` does not conform to the `NSValidatedUserInterfaceItem` protocol. The New Chef's Diary/Open Chef's Diary menu item is different, in that it does not have a corresponding control to perform the same operation. Menu items can be validated in a `-validateUserInterfaceItem:` protocol method without creating a custom protocol, because `NSMenuItem`, unlike `NSControl`, conforms to the `NSValidatedUserInterfaceItem` protocol.

The correct place to implement the protocol method for this menu item is in the `VRDocumentController` class, which manages all of the application's documents and implements the menu item's action methods. The `NSDocumentController` base class already conforms to the `NSUserInterfaceValidations` protocol by implementing `-validateUserInterfaceItem:` for the New Document, Open Document, and Open Recent menu items, among others. You override it here to validate the New Chef's Diary/Open Chef's Diary menu item.

Start with a simple, preliminary version of the protocol method, taking account only of the possibility that a diary window is already open and has not yet been saved. In the `VRDocumentController.m` implementation file, implement this method, including the `#pragma mark` statement immediately above it:

```
#pragma mark USER INTERFACE VALIDATION
```

```
- (BOOL)validateUserInterfaceItem:
    (id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if ((action == @selector(newDiaryDocument:))
        || (action == @selector(openDiaryDocument:))) {
        for (NSDocument *thisDocument in [self documents]) {
            if ([thisDocument isKindOfClass:[DiaryDocument class]]) {
                return NO;
            }
        }
        return YES;
    }
    return [super validateUserInterfaceItem:item];
}
```

Remember that the `-validateUserInterfaceItem:` method is called repeatedly, once for each of a menu's menu items, whenever the user opens one of the application's menus. It is called just before the menu is displayed, so there is still time to change the appearance or even the title of the menu items. This simple version of the `-validateUserInterfaceItem:` method first gets the action of the menu item that was passed in the `item` parameter, and then it tests whether the item's action is the `newDiaryDocument:` or the `openDiaryDocument:` selector. It doesn't matter that you haven't yet implemented the `-openDiaryDocument:` method; that branch of the test is just evaluated as false. If the action satisfies either of these tests, the method checks the document controller's array of open documents by calling its `-documents` method, testing whether the class of any of the open documents is `DiaryDocument` or a subclass of `DiaryDocument`. If so, the method returns `NO`, disabling the menu item so that the user cannot create another new diary document; if not, it returns `YES`, enabling the menu item. If the current item is not the menu item of interest, the method calls `[super`

`validateUserInterfaceItem:item]` so that the `NSDocumentController` base class can validate the menu items for which it is responsible.

You may recall that the first time you implemented the `-validateUserInterfaceItem:` protocol method, you returned YES at the end instead of letting the superclass perform its validation, because in that case the superclass did not implement the protocol method. It is important to return the superclass's implementation in your override of the protocol method if the documentation indicates that it implements that method. Otherwise, the menu items normally validated by the superclass will not be validated.

If you build and run the application now and choose File > New Chef's Diary, you find that the menu item is enabled. Choose it, and a Chef's Diary document opens. Now try to choose File > New Chef's Diary again, and you find that the menu item is disabled. Close the document without saving it and choose File > New Chef's Diary again, and the menu item is once again enabled.

Later in this step, you will return to this protocol method and revise it to handle the case where the open diary window reflects a diary document that the user has already saved. First, however, you must tend to the methods required to allow the user to save it.

2. You will shortly create a method that writes a value to the user defaults when the user saves a new diary document, so that the application can remember that the user has in fact created and saved a diary document. This information is required to enable the application to decide whether to change the title of the New Chef's Diary menu item to *Open Chef's Diary* and to change its action to the `openDiaryDocument:` selector.

The value that the application writes to user defaults should be a reference to the saved file in a form that allows the application to locate and reopen it when the user chooses File > Open Chef's Diary. The application should present a dialog asking the user to find the file only if the application is unsuccessful in locating it through the user defaults value. To maximize the chance that the application can find the file even if the user has moved or renamed it, write the value to user defaults as an alias record in Leopard or as a bookmark in Snow Leopard. Store `nil` in the user defaults, or remove the value, if the Chef's Diary can't be found and the user responds to the dialog by canceling, so that the menu item's title reverts to *New Chef's Diary* the next time the user opens the File menu.

The first step toward creating this mechanism is to write a method that converts a standard file URL to an `NSData` object representing, in Leopard, an alias record or, in Snow Leopard, a bookmark. It is customary to use an `NSData` object for this purpose, because it is very easy to store and retrieve `NSData` objects. The application uses the method to write the `NSData` object to user defaults. Later, for the application's use after it retrieves the alias record or bookmark from user defaults, you will write a method to convert the `NSData` object back to a file URL.

suitable for opening the document. The application will test whether the user defaults value is `nil` to decide how to set the current title of the menu item and its current action.

If you planned to use this mechanism only for the Chef's Diary, you could implement it in the `DiaryDocument` class. In `Vermont Recipes`, however, you haven't yet specified how the main recipes document will be handled, and you might want to use the same mechanism for it. These are general-purpose methods. You should therefore implement them in `VRDocumentController`, where they are available to the application at all times through the `+sharedDocumentController` class method. You could even implement them in a separate class or a category on `NSURL`, if you thought you might have use for them in another application.

Declare the `-aliasDataFromURL:` method at the end of the `VRDocumentController.h` header file like this:

```
#pragma mark ALIAS/BOOKMARK MANAGEMENT
```

```
- (NSData *)aliasDataFromURL:(NSURL *)fileURL;
```

Define it in the `VRDocumentController.m` implementation file like this:

```
#pragma mark ALIAS/BOOKMARK MANAGEMENT
```

```
- (NSData *)aliasDataFromURL:(NSURL *)fileURL {
    if (fileURL == nil) return nil;
    if (floor(NSFoundationVersionNumber)
        > NSFoundationVersionNumber10_5) {
        return [fileURL bookmarkDataWithOptions:0
            includingResourceValuesForKeys:nil
            relativeToURL:nil error:NULL];
    } else {
        FSRef fsRef;
        if (CFURLGetFSRef((CFURLRef)fileURL, &fsRef)) {
            AliasHandle aliasHdl;
            OSErr err = FSNewAlias(NULL, &fsRef, &aliasHdl);
            if ((err == noErr) && (aliasHdl != NULL)) {
                NSData *returnData = [NSData dataWithBytes:*aliasHdl
                    length:GetAliasSize(aliasHdl)];
                DisposeHandle((Handle)aliasHdl);
                return returnData;
            }
        }
        return nil;
    }
}
```

In Mac OS X 10.6 Snow Leopard, Apple has made a substantial push to complete its longstanding program to modernize Cocoa file handling by standardizing on the `NSURL` class. As part of this process, it has supplemented the traditional Carbon Alias Manager with a new Cocoa bookmark API that serves the same purpose in a more accessible and efficient manner. Among other things, it has added to `NSURL` a number of methods to create and resolve bookmarks. The `-aliasDataFromURL:` method uses these new Snow Leopard bookmark facilities to create a bookmark-based `NSData` object if Vermont Recipes is running under Snow Leopard. To detect whether the application is running under Snow Leopard, the method tests `NSFoundationVersionNumber` to see if it is greater than `NSFoundationVersionNumber10_5`. This is similar to the `NSAppKitVersionNumber` test you first used in Recipe 2.

The `NSURL` bookmark methods include the ability to save and retrieve various file system options and values much more efficiently than with the Carbon Alias Manager, but you don't need them here, so you pass values of `0` and `nil` in the first two parameters. You also pass `nil` in the parameter labeled `relativeToURL:` because you want to write an absolute URL to user defaults, not a relative URL. Pass `NULL` in the `error:` parameter for now; you will deal with the `error:` parameter found in many file management methods in the next step.

If Vermont Recipes is running under Mac OS X 10.5 Leopard, the `-aliasDataFromURL:` method must use the Carbon Alias Manager and the Core Foundation `FSRef` type. Until the advent of Snow Leopard, Cocoa had no convenient facilities of its own to create or resolve alias records or alias files, so by necessity Cocoa developers have always resorted to the Carbon Alias Manager to deal with alias records and alias files.

For a detailed understanding of the old way to create and resolve alias records and alias files, read the several Apple documents on this topic collected in the Documentation sidebar at the end of this recipe. This knowledge will remain relevant to Cocoa developers as long as they continue to support Leopard, because Snow Leopard's bookmark methods do not know how to read or write old-style Alias Manager alias records.

In summary, if the application is running under Leopard, the `-aliasDataFromURL:` method calls the Core Foundation function `CFURLGetFSRef()` to convert the `fileURL` parameter value to an `FSRef`. The `CFURL` and `NSURL` data types are toll-free bridged, so you can use them interchangeably with appropriate type casting. The method then calls the Alias Manager's `FSNewAlias()` function to convert the `FSRef` to an `AliasHandle`. Finally, it uses `NSData`'s `+dataWithBytes:length:` method to place the `AliasHandle`'s data into an `NSData` object, which it returns for use by the methods you will write shortly to store it in user defaults. When it is finished with the `AliasHandle`, it calls `DisposeHandle()` for proper memory management.

Note that the term *alias*, rather than the term *bookmark*, is used in the name of the `-aliasDataFromURL:` method. The new NSURL bookmark feature is intended to replicate the functionality that Mac users have long associated with aliases, and it seems reasonable to anticipate that they will continue to be called aliases in everyday developer jargon.

3. Write the corresponding method to convert the alias record or bookmark data back to a file URL now, while you're still thinking about aliases and bookmarks.

Bear in mind that if the application is running under Snow Leopard, this method must be able to convert both old-style Alias Manager alias record data and Snow Leopard bookmark data to a valid URL. The user may have saved the application's user defaults while running under Leopard and then upgraded the computer to Snow Leopard. When the user runs the application under Snow Leopard, it must be capable of reading the data from user defaults whether it is in the form of an Alias Manager alias record or a bookmark.

Declare the `-URLFromAliasData:` method at the end of the `VRDocumentController.h` header file like this:

```
#pragma mark ALIAS/BOOKMARK MANAGEMENT
```

```
- (NSURL *)URLFromAliasData:(NSData *)aliasData;
```

Define it in the `VRDocumentController.m` implementation file like this:

```
- (NSURL *)URLFromAliasData:(NSData *)aliasData {
    if (aliasData == nil) return nil;
    NSURL *returnURL = nil;
    if (floor(NSFoundationVersionNumber)
        > NSFoundationVersionNumber10_5) {
        BOOL isStale;
        returnURL = [NSURL URLByResolvingBookmarkData:aliasData
                    options:0 relativeToURL:nil
                    bookmarkDataIsStale:&isStale error:NULL];
        if (returnURL != nil) {
            return returnURL;
        }
    }

    AliasHandle aliasHdl = NULL;
    PtrToHand([aliasData bytes], (Handle*)&aliasHdl, [aliasData length]);
    if (aliasHdl != NULL) {
        FSRef fsRef;
        Boolean wasChanged;
```

```

        OSErr err = FSResolveAlias(NULL, aliasHdl, &fsRef, &wasChanged);
        DisposeHandle((Handle)aliasHdl);
        if (err == noErr) {
            returnUrl = [(NSURL *)CFURLCreateFromFSRef(NULL, &fsRef)
                          autorelease];
        }
    }
    return returnUrl;
}

```

The `-URLFromAliasData:` method first checks whether Snow Leopard is running. If so, it calls the new Snow Leopard NSURL method, `+URLByResolvingBookmarkData:options:relativeToURL:error:`, on the assumption that the `aliasData` parameter holds Snow Leopard bookmark data. If this successfully converts the data and returns a URL, the method returns it and is done. If Snow Leopard is not running, or if the Snow Leopard method returns `nil` because the `aliasData` parameter held Carbon Alias Manager alias record data, the `-URLFromAliasData:` method converts it to a URL using old-style Alias Manager techniques and returns the result. The method returns `nil` if nothing succeeds in producing a URL.

The result of the `CFURLCreateFromFSRef()` function is returned as an autoreleased NSURL object. This is necessary because the function includes the term *create*, and, as its documentation states, it therefore follows the Core Foundation *Create rule*. This rule provides that you own the returned Core Foundation object, a CFURLRef object, and you are therefore responsible for releasing it. Because CFURL is toll-free bridged with NSURL, you are free to use Cocoa's `autorelease` method if you cast the returned CFURLRef to an NSURL object.

Both the Leopard function and the Snow Leopard method return by reference a value telling you whether the alias record or bookmark needs to be updated because, for example, the user has moved or renamed the original item to which it points. In Leopard, this is the `wasChanged` parameter value; in Snow Leopard, it is the `isStale` parameter value. The `-URLFromAliasData:` method ignores both of these parameter values because the application uses it only as a conversion method. Instead, shortly, when you write methods to open an existing diary document, you will compare the URL returned by the `-URLFromAliasData:` method with the URL of the object that the user attempts to open, and if they differ, you will update the user defaults to reflect the new location of the diary document.

4. Next, determine the file URL to which the user saved the diary document, convert it to an alias record or bookmark data using your `-aliasDataFromURL:` method, and write the alias record or bookmark to the user defaults.

To learn where the user saved the document, you have to take into account the strategy that `NSDocument` follows when saving a document. This strategy is laid out in detail in the “Saving a Document” subsection of the “Message Flow in the Document Architecture” section of Apple’s *Document-Based Applications Overview*. It is important to know that `NSDocument` sometimes writes a document to a temporary location and then moves it to its final location. Although several of the methods that `NSDocument` calls during this process take a file URL as a parameter, you must be careful to work with one of them that uses the final file URL, not the initial temporary file URL. Also, you must work with a method that is always called for the three save operation types used by `NSDocument` to save a file in its final location—namely, `NSSaveOperation`, `NSSaveAsOperation`, and `NSSaveToOperation`.

A method that meets all these requirements is `-[NSDocument saveToURL:ofType:forSaveOperation:error:]`. It is always called late in the process of saving a document. Override it now, using its `absoluteURL` parameter to create the alias record or bookmark, and save it to user defaults.

In the `DiaryDocument.m` implementation file, insert this override method implementation just after your override of the `-dataOfType:error:` method:

```
- (BOOL)saveToURL:(NSURL *)absoluteURL ofType:(NSString *)typeName
    forSaveOperation:(NSSaveOperationType)saveOperation
    error:(NSError **)outError {
    BOOL success = [super saveToURL:absoluteURL ofType:typeName
                        forSaveOperation:saveOperation error:outError];
    if (success && (saveOperation != NSAutosaveOperation)) {
        NSData *diaryAlias =
            [[VRDocumentController sharedDocumentController]
             aliasDataFromURL:absoluteURL];
        [[NSUserDefaults standardUserDefaults] setObject:diaryAlias
            forKey:VRDefaultDiaryDocumentAliasDataKey];
    }
    return success;
}
```

The method first calls super’s version of the same method, which ensures that the `NSDocument` strategy for writing data to disk actually writes the diary’s data to disk. You then scavenge information from the incoming parameters, which the `NSDocument` mechanism has already set up. This is a common Cocoa design pattern; override one of the Cocoa framework’s methods that you know Cocoa will call at an appropriate time, call super’s version of the same method to make sure it does its job, and then use the parameter data supplied by the framework for your own purposes.

Here, if the save operation was successful, the override method first tests whether the save operation was not an autosave operation, because you don't need to update user defaults as a result of an autosave operation. If the save operation was one of the other three types of save operations, the method calls the `-aliasDataFromURL:` method that you just wrote, converting the `absoluteURL` parameter value, to which `NSDocument` wrote the diary data, into an `NSData` object suitable for storing in user defaults.

Recall that you originally placed the `-aliasDataFromURL:` method in `VRDocumentController` because you might want to use it for the recipes document as well as the diary document. Because it is declared in `VRDocumentController`, you must import `VRDocumentController`. Near the top of the `DiaryDocument.m` implementation file, after the other `#import` directives, insert this line:

```
#import "VRDocumentController.h"
```

Finally, the override method writes the `NSData` object to the user defaults. Every application's user defaults are accessible through the `NSUserDefaults` shared singleton object, using its `+standardUserDefaults` class method. You will explore Cocoa's user defaults mechanism in detail later. For now, you only need to know that it stores values and associated keys in a dictionary-like object. Here, the object to be written to user defaults is the `NSData` object encoding the alias record or bookmark for the saved diary document. The corresponding key is the value held by the `VRDefaultDiaryDocumentAliasDataKey` variable. By setting the object in `standardUserDefaults` with that key, you ensure that Cocoa writes it to disk at a suitable time.

But where did the `VRDefaultDiaryDocumentAliasDataKey` variable come from? You have yet to create it. You could simply define it using a `#define` directive, but then it would be available only in the file where you defined it, unless you took steps to make it available everywhere it is needed. Instead, you will follow a common Cocoa technique to declare and implement global `NSString` and other variables used in multiple files. This variable is needed both in the `DiaryDocument` class and in the `VRDocumentController` class. Declare and implement it in `VRDocumentController`.

In the `VRDocumentController.h` header file, just above the `@interface` directive, declare the global variable like this:

```
extern NSString *VRDefaultDiaryDocumentAliasDataKey;
```

At the end of the `VRDocumentController.m` implementation file, following the `@end` directive, define it like this:

```
NSString *VRDefaultDiaryDocumentAliasDataKey =  
    @"diary document alias data";
```

5. You are going to read and write the diary document's URL in the user defaults with some frequency, so it would be a good idea to create a pair of methods to do that. Call them `-currentDiaryURL` and `-setCurrentDiaryURL:`. These look like accessor methods, and they are accessor methods. You may normally think of accessor methods as accessing instance variables or properties of an object, but that need not always be the case.

In the `VRDocumentController.h` header file, declare them like this at the beginning of the Alias/Bookmark Management section:

```
- (NSURL *)currentDiaryURL;  
- (void)setCurrentDiaryURL:(NSURL *)absoluteURL;
```

Define them in the `VRDocumentController.m` implementation file:

```
- (NSURL *)currentDiaryURL {  
    NSData *aliasData = [[NSUserDefaults standardUserDefaults]  
        objectForKey:VRDefaultDiaryDocumentAliasDataKey];  
    return [self URLFromAliasData:aliasData];  
}  
- (void)setCurrentDiaryURL:(NSURL *)absoluteURL {  
    NSData *aliasData = [self aliasDataFromURL:absoluteURL];  
    [[NSUserDefaults standardUserDefaults] setObject:aliasData  
        forKey:VRDefaultDiaryDocumentAliasDataKey];  
}
```

The `-currentDiaryURL` getter method gets the document's alias data from the user defaults and then converts it to a URL using the method you wrote earlier while you were thinking about aliases and bookmarks, `-URLFromAliasData:`.

Now go back to the `-saveToURL:ofType:forFileOperation:error:` method that you just wrote and use your new `-setCurrentDiaryURL:` setter method. Replace the two statements in the `if` block with this one statement:

```
[[VRDocumentController sharedDocumentController]  
    setCurrentDiaryURL:absoluteURL];
```

6. Now that the application has a record of where and under what name the user saved the diary document, you should allow the user to open it using that information any time some new culinary experience requires annotation. Start by writing the `-openDiaryDocument:` action method that has already been mentioned a number of times; then arrange to change the New Chef's Diary menu item's title to Open Chef's Diary and change its action to `-openDiaryDocument:`.

In the `VRDocumentController.h` header file, enter this declaration following the existing `-newDiaryDocument:` declaration:

```
- (IBAction)openDiaryDocument:(id)sender;
```

In the `VRDocumentController.m` implementation file, enter this definition:

```
- (IBAction)openDiaryDocument:(id)sender {
    NSDocument *diaryDocument = [self diaryDocument];
    if (diaryDocument != nil) {
        [[[diaryDocument windowControllers] objectAtIndex:0] window]
            makeKeyAndOrderFront:sender];
    } else {
        NSURL *fileURL = [self currentDiaryURL];
        if (fileURL != nil) {
            DiaryDocument *diary =
                [self makeDocumentWithContentsOfURL:fileURL
                 ofType:DIARY_DOCUMENT_IDENTIFIER error:NULL];
            // FIXME: Add error handling.
            if (diary != nil) {
                [self addDocument:diary];
                [diary makeWindowControllers];
                [diary showWindows];
            }
        }
    }
}
```

The `-openDiaryDocument:` action method first attempts to get the Chef's Diary document on the assumption that it is already open. It does this by calling `-diaryDocument`, a simple method that you will write shortly to iterate over the document controller's list of open documents. If it is open, the return value is the `diaryDocument` object, not `nil`, and in that case the method simply brings the document's window to the front using the very useful `NSWindow` method `-makeKeyAndOrderFront:`, which does just what its name suggests.

Otherwise, the action method attempts to get the document's URL using the `-currentDiaryURL` getter method you wrote earlier. If the result is not `nil`, it creates the document object from the contents of the URL with `NSDocumentController`'s `-makeDocumentWithContentsOfURL:ofType:error:` method; then it adds the document object to the document controller's list of open documents, makes its window controller, adds the controller to its internal list of window controllers, and opens the window.

7. Write the `-diaryDocument` method, mentioned earlier, to iterate over the document controller's list of open documents. In the `VRDocumentController.h` header file, declare it:

```
#pragma mark UTILITY METHODS

- (NSDocument *)diaryDocument;
```

Implement it in the `VRDocumentController.m` implementation file:

```
#pragma mark UTILITY METHODS
```

```
- (NSDocument *)diaryDocument {
    for (NSDocument *thisDocument in [self documents]) {
        if ([thisDocument isKindOfClass:[DiaryDocument class]]) {
            return thisDocument;
        }
    }
    return nil;
}
```

This method simply encapsulates the `for` loop in a separate method, so the operation can be performed with a simple method call in several other methods. You will use it again shortly when you revise the `-validateUserInterfaceItem:` method.

8. With both the `-newDiaryDocument:` and the `-openDiaryDocument:` action methods implemented, you can assign the appropriate title and action to the menu item. Both the title and the action of the menu item depend on whether the application's user defaults return `nil` for the diary document alias entry. If there is no alias record or bookmark in user defaults and attempting to retrieve it therefore returns `nil`, the menu item's title should be *New Chef's Diary* and its action selector should be `newDiaryDocument:`. If there is an entry, the title should be changed to *Open Chef's Diary* and the action selector should be changed to `openDiaryDocument:`.

Both the title and the action can be set in the `-validateUserInterfaceItem:` method you wrote earlier in this step. Validation is not limited to enabling or disabling menu items. It can make any changes to the menu based on the application's state at the moment the menu is opened.

Delete the existing `-validateUserInterfaceItem:` method implementation in the `VRDocumentController.m` implementation file, and substitute the following new version:

```
- (BOOL)validateUserInterfaceItem:
    (id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if ((action == @selector(newDiaryDocument:))
        || (action == @selector(openDiaryDocument:))) {

        NSURL *fileURL = [self currentDiaryURL];
        if ((fileURL != nil)
            && [self canOpenURL:fileURL]) {
```

```

        [(NSMenuItem *)item setTitle:
            NSLocalizedString(@"Open Chef's Diary",
                @"menu item title for Open Chef's Diary")];
        [(NSMenuItem *)item setAction:@selector(openDiaryDocument:));
    } else {
        [(NSMenuItem *)item setTitle:
            NSLocalizedString(@"New Chef's Diary",
                @"menu item title for New Chef's Diary")];
        [(NSMenuItem *)item setAction:@selector(newDiaryDocument:));
    }

    return (([self diaryDocument] == nil)
        || ([NSMenuItem *)item action]
            == @selector(openDiaryDocument:)));
}
return [super validateUserInterfaceItem:item];
}

```

The new version of `-validateUserInterfaceItem:` changes the `if` block to supplement the strategy followed by the original version. It tests not only whether the document is already open but also whether it exists and can be opened. The new version of the `if` block starts by attempting to get the URL for the Chef's Diary document from the user defaults using the `-currentDiaryURL` getter method. Then it calls a method you have yet to write, `-canOpenURL:`, to determine whether it exists and is not in the Trash. If so, it sets the title of the menu item to *Open Chef's Diary* and its action selector to `openDiaryDocument:`. If not, it sets the menu item's title to *New Chef's Diary* and its action selector to `newDiaryDocument:`. Because you have set the actions programmatically, there is no need to connect them in Interface Builder.

Finally, it enables or disables the menu item by returning `YES` or `NO` as appropriate. It does this, in part, by calling the `-diaryDocument` method you just wrote to determine whether the document is already open. It performed the same test in the original version, but there the `for` loop was inline. The new version also tests whether the menu item's title is now *Open Chef's Diary*.

Here's the logic for enabling or disabling the menu item: If no Chef's Diary document is open—that is, `-diaryDocument` returns `nil`—or if a Chef's Diary document exists—that is, the action selector just assigned to the menu item is `openDiaryDocument:`—it returns `YES` to enable the menu item so that the user can either create a new document or open or activate a saved document. If both conditions are false—that is, a Chef's Diary document is already open and it has not yet been saved—it returns `NO` to disable the menu item so that the user cannot create a new one.

9. Now write the `-canOpenURL:` method you just used. In the `VRDocument-Controller.h` header file, declare it like this:

```
- (BOOL)canOpenURL:(NSURL *)absoluteURL;
```

In the implementation file, implement it like this:

```
- (BOOL)canOpenURL:(NSURL *)absoluteURL {
    if (absoluteURL != nil) {
        Boolean isInTrash = true;
        DetermineIfPathIsEnclosedByFolder
            (kOnAppropriateDisk, kTrashFolderType,
             (const UInt8 *)[NSFileManager defaultManager]
             fileSystemRepresentationWithPath:[absoluteURL path]],
             false, &isInTrash);
        if (floor(NSFoundationVersionNumber)
            <= NSFoundationVersionNumber10_5) {
            return [[NSFileManager defaultManager]
                    fileExistsAtPath:[absoluteURL path]] && !isInTrash;
        } else {
            return [absoluteURL
                    checkResourceIsReachableAndReturnError:NULL]
                    && !isInTrash;
        }
    }
    return NO;
}
```

This method is needed primarily to deal with the case in which the original Chef's Diary document exists and has been saved but is currently in the Trash. Vermont Recipes takes this to mean that the user is through with it and intends to create a new, empty Chef's Diary. Although the user can recant by removing the file from the Trash, as long as it is in the Trash, the application should treat it as nonexistent.

Unfortunately, Cocoa file-handling routines are not so discriminating. They happily report the existence of a file even if it is in the Trash. If you research the question of how to find out whether a file is in the Trash, you will be surprised to discover that Cocoa does not know how to do this, even in Snow Leopard. Instead, you have to call an obscure Core Services function introduced in Mac OS X 10.4 Tiger, which, judging from the mailing lists, hardly anybody knows about: `DetermineIfPathIsEnclosedByFolder()`. It is, however, documented in the Folder Manager Reference, which includes a snippet of code showing how to tell whether a file is in the Trash. The `-canOpenURL:` method uses this function to set the `isInTrash` local variable.

Then, depending on whether it is running on Leopard or Snow Leopard, the `-canOpenURL:` method calls an appropriate method to determine whether the file exists, returning YES only if the file exists and is not in the Trash. In Leopard, it calls `NSFileManager`'s `-fileExistsAtPath:` method; in Snow Leopard, it calls `NSURL`'s new `-checkResourceIsReachableAndReturnError:` method.

10. There remains one final method to write. You already wrote the `-openDiaryDocument:` action method, which is called when the user chooses the Open Chef's Diary menu item. However, there are several other ways to open documents, such as choosing the Open and Open Recent menu items in the application's File menu, choosing the Finder's File > Open menu item, double-clicking the file's icon in the Finder, and dropping the file on the application's icon in the Finder or the Dock. All of these techniques end up calling `NSDocumentController`'s `-openDocumentWithContentsOfURL:display:error:` method, some of them through `NSDocumentController`'s `-openDocument:` action method and some by sending the open document ('`odoc`') Apple event when the user opens the document in the Finder.

As its documentation indicates, by default the `-openDocumentWithContentsOfURL:display:error:` method does several of the things that you did yourself in the `-openDiaryDocument:` action method, such as calling `-makeDocumentWithContentsOfURL:ofType:error:`, `-addDocument:`, `-makeWindowControllers`, and `-showWindows`. However, just as you did in the action method, you need to implement the specification calling for only one diary document whenever the user uses these other techniques to open it, such as double-clicking its icon in the Finder.

Do this by overriding `-openDocumentWithContentsOfURL:display:error:`. In the `VRDocumentController.m` implementation file, implement this override method:

```
- (id)openDocumentWithContentsOfURL:(NSURL *)absoluteURL
    display:(BOOL)displayDocument error:(NSError **)outError {
    NSDocument *diaryDocument = [self documentForURL:absoluteURL];
    if (diaryDocument != nil) {
        [[[[diaryDocument windowControllers] objectAtIndex:0] window]
         makeKeyAndOrderFront:nil];
        return diaryDocument;
    } else if ([self diaryDocument] != nil) {
        if (outError != NULL) {
            *outError = [NSError errorWithDomain:NSCocoaErrorDomain
                                           code:NSUserCancelledError userInfo:nil];
            // TODO: add error handling.
        }
    }
```

(code continues on next page)

```

    } else {
        NSURL *currentDiaryURL = [self currentDiaryURL];
        if ((currentDiaryURL == nil)
            || [absoluteURL isEqual:currentDiaryURL]) {
            diaryDocument =
                [super openDocumentWithContentsOfURL:absoluteURL
                 display:displayDocument error:outError];
            if (diaryDocument != nil) {
                if (currentDiaryURL == nil)
                    [self setCurrentDiaryURL:absoluteURL];
                return diaryDocument;
            }
        } else {
            if (outError != NULL) {
                *outError = [NSError errorWithDomain:NSCocoaErrorDomain
                 code:NSUserCancelledError userInfo:nil];
                // TODO: add error handling.
            }
        }
    }
    return nil;
}

```

The method is fairly complex, because it has to deal with several possibilities. If the document that the user is trying to open is already open, you should just bring it to the front and return it as the method's return value. If a different document is already open, you should not open the document that the user is now trying to open, both because Vermont Recipes allows only one diary document to be open at a time and because the new document isn't the document designated in the user defaults as the current document (you know it isn't the current document because the document that is already open is, by definition, the current diary document). You should therefore ignore the attempt or report an error. If no diary document is currently open, you should open it if it is the application's current diary document or the application does not yet have a current diary document. In the latter case, you must also save an alias record or bookmark for it in user defaults so that it will hereafter be treated as the current diary document. If the document that the user is trying to open is different from the current diary document, you must again do nothing or report an error.

To accomplish all this, the method first checks whether the document that the user is trying to open, at `absoluteURL`, is already open. It calls `NSDocumentController`'s built-in `-documentForURL:` method, which is designed for just this purpose. If it is already open, the method simply brings its window to the front and returns the document object as the method's return value.

If the document that the user is trying to open is not already open, the method next checks whether any diary document is already open, by calling the `-diaryDocument` method you wrote earlier in this step. If another diary document is already open, the method does something with the `outError` argument that I'll explain in a moment. Then it falls through to the end of the method and returns `nil` to signal that an error has occurred.

Finally, if no document is already open, the method checks whether the application has not yet saved a reference to its current diary document or, if it has, whether the document that the user is trying to open is the current diary document. In either case, it calls the superclass's implementation to open the document, and it returns the document as the method's return value. Along the way, if the application does not yet have a current diary document, the method saves a reference to this document as the current diary document. If the document that the user is trying to open is different from the current diary document, the method again does something with the `outError` argument and returns `nil`.

The interesting action comes in the two error situations, where a diary document is already open or the user tries to open a document that is different from the current diary document. For now, the method generates an `NSError` object indicating that the user canceled the open operation, and it returns `nil` to indicate that it has not opened the document. This hardly makes for a friendly user experience, however, so you should handle the error more appropriately. In the next step, where you explore Cocoa error handling in some depth, you will arrange to present a dialog informing the user of the error and giving the user an opportunity to deal with it.

For now, leave the error handling as you see it here. In both cases, you first test whether the `outError` parameter value is `NULL`, and if it is not, you set it to an `NSError` object having the `NSCocoaErrorDomain` domain and the `NSUserCancelledError` error code. This is the technique that Apple recommends you use whenever you wish to suppress an error alert and ignore the error. In fact, this very code snippet appears in the "Error Handling in the Document Architecture" section of the *Document-Based Applications Overview*.

You have now implemented a flexible mechanism to let the user save the one and only Chef's Diary document under any name and in any location, while still allowing it to be reopened from a single menu item even after the user has moved and renamed it. Thanks to aliases and bookmarks, it is not necessary to hide one-of-a-kind documents in obscure locations like `~/Library/Application Support`.

Before trying it out, drag any existing Chef's Diary document that you already created to the Trash. Now open Vermont Recipes' File menu. You see that the second menu item is New Chef's Diary, and it is enabled. Choose File > New Chef's Diary. A new Chef's Diary document is created and its window opens. The window's title

is Untitled, followed by a number if you've done this more than once, courtesy of functionality built into document-based applications. Type some placeholder text and choose File > Save As. In the Save panel, type **One** as the name of the file, choose the desktop as its location, and click Save. The document icon appears on the desktop with the name *One* that you just gave it. Use the Finder's Get Info command on it, and deselect the "Hide extension" setting. Now you see the file's name with its file extension, *One.vrdiary*. The window's title is now *One*, the name you just gave to the document, again courtesy of built-in document-based application functionality.

Open the File menu again. You see that the second menu item is now Open Chef's Diary, and it is enabled. Click the main Vermont Recipes window to bring it to the front and cause the diary window to move to the back. Now choose File > Open Chef's Diary, and the diary document's window comes to the front. Close the diary document and choose File > Open Chef's Diary again. The diary document reopens. Close it again and double-click its icon on the desktop, and it reopens again.

Now comes the fun part. First, close the diary document. In the Finder, rename it *Two* and drag it into, say, your Pictures folder. In Vermont Recipes, choose File > Open Chef's Diary, and it reopens, just as you expected, and the window is correctly titled *Two*. Close it again, and drag the file from the Pictures folder to the Trash, but don't empty the Trash. Now when you open the Vermont Recipes File menu, the second menu item is New Chef's Diary. Drag the file back out of the Trash onto the desktop, and you can once again open it by choosing File > Open Chef's Diary. Open it now.

Next, with the diary document's window open, choose File > Save As and save the open diary document on the desktop under yet another name, *Three*. Close the diary document, and then choose File > Open Chef's Diary. This time, the new document you just created, *Three.vrdiary*, opens. It has now been specified as the current diary document in user defaults. Close its window and double-click its icon on the desktop, and it reopens as expected.

You can treat the old version of the diary document, *Two.vrdiary*, as a backup or archive file. But if for any reason you lose the newer *Three.vrdiary* file or become unhappy with it, how will you go back to using *Two.vrdiary* as the current diary document? There is no way to open it from Vermont Recipes, because the Open Chef's Diary menu item opens the new *Three.vrdiary* file. Try double-clicking the *Two.vrdiary* document's icon in the Finder. You see the Finder's opening document animation and Vermont Recipes activates, but the *Two.vrdiary* does not open. Fortunately, you anticipated this behavior, and you already inserted a **TODO:** comment in the code to remind you that this requires more work. You will attend to it in the next step by bringing up a dialog asking the user whether to make the older diary document the current Chef's Diary document. Eventually, in a later recipe, you will even set up the application's preferences so that the user can specify any diary document created by Vermont Recipes, and perhaps any existing RTF file, as the current Chef's Diary document.

Step 3: Add Error Handling to the Diary Document

At the end of Step 2, you encountered two situations that might appropriately be considered errors. The user tried to open an existing diary document while another diary document was already open, or a user tried to open an existing diary document that was not the application's current diary document. Some sort of error message is appropriate in both cases to alert the user to the nature of the problem. Silent failure is not an option in a user-friendly application.

Interestingly, the user could turn the second error to advantage if you were to provide some helpful code. The user has double-clicked an existing Chef's Diary document or dropped it on the Vermont Recipes document icon, but Vermont Recipes can't open it because the user defaults say that another diary document is the current Chef's Diary document. The situation cries out for a dialog offering the user an opportunity to substitute the old diary document's URL for the new document's URL in user defaults, effectively switching back to the backup as the current Chef's Diary document. Alternatively, the dialog could offer to open the correct diary document for the user.

Both situations are perfect opportunities to use Cocoa's `NSError` class. At the moment, the `-openDocumentWithContentsOfURL:display:error:` method contains stopgap error handling code that pretends the user canceled the attempt to open the offending diary document. You should now change this to provide a more meaningful error message as well as a mechanism, in the second situation, for the user to change the current Chef's Diary document.

1. For a first try, simply change the error domain and the error code in both calls to `+errorWithDomain:code:userInfo:` in the `-openDocumentWithContentsOfURL:display:error:` method you wrote at the end of Step 2. Currently, they set `outError` to an `NSError` object with the `NSCocoaErrorDomain` error domain and the `NSUserCancelledError` error code. As you learned in Step 2, these settings prevent the application from presenting any error alert. To see the alert that would otherwise be presented, change the error domain to `NSURLErrorDomain` and the error code to `NSURLErrorCannotOpenFile`, so that both statements read as follows:

```
*outError = [NSError errorWithDomain:NSURLErrorDomain
                           code:NSURLErrorCannotOpenFile userInfo:nil];
```

Now build and run the application. Trash any existing diary documents and choose `File > New Chef's Diary` to create a new one. To test the new error-handling code, choose `File > Save As` to save the document as *Backup.vrdiary*, and then choose `File > Save As` to save it again as *New.vrdiary*. *New.vrdiary* is now the diary document specified in user defaults as the current Chef's Diary.

Double-click the Backup.vrdiary file icon on the desktop. This time, instead of nothing happening, an error alert opens with an error icon, an OK button, and text reading “The document ‘Backup.vrdiary’ could not be opened” (**Figure 6.3**). When you click OK, the alert is dismissed, and the application waits for your next instruction.

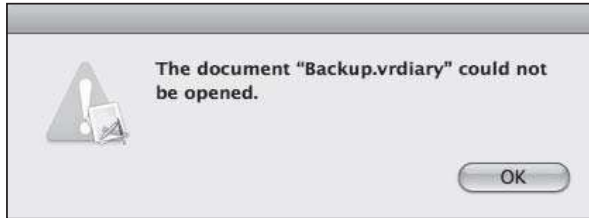


FIGURE 6.3 The default error alert presented when the user attempts to open a backup Chef’s Diary document.

If you do nothing more than this to handle errors in a document-based application, the application takes care of communication with the user for you. If you can find a suitable preexisting error code like `NSURLErrorCannotOpenFile`, use it and its error domain as you did here, and Cocoa will present a reasonable localized error alert. The generic Cocoa error domain is `NSCocoaErrorDomain`, and generic error codes in that domain are listed in a handful of header files, `AppKitErrors.h` in the AppKit framework, `FoundationErrors.h` in the Foundation framework, and `CoreDataErrors.h` in the Core Data framework. You will find these frameworks and their headers in `/System/Library/Frameworks`.

There are a few more specialized error domains. The one you used here is `NSURLErrorDomain`, whose error codes are listed in the `NSURLError.h` header file in the Foundation framework. For document-handling errors in Snow Leopard, with its emphasis on file URLs, `NSURLErrorDomain` is the first place you should look, but be sure to check `FoundationErrors.h`, too.

You can even define your own application-specific error domains and error codes, but there is no need to go that far here.

2. The simple error message supplied by Cocoa is not very informative. It doesn’t explain why the application could not open the document, and it doesn’t give a clue as to what the user can do about it.

To provide a more informative error message, create your own text. In general, you provide custom error messages to the user in two steps. First, you create your own customized `NSError` object containing the text and other user interface features you desire. Second, you implement a method like `NSDocumentController`’s `-willPresentError:` to present your error instead of the error that Cocoa presents by default.

`NSError` is designed to be very flexible, and there are many ways you can do this. Here, you will use a relatively simple technique. In more complex applications, you might be better off redesigning this approach to make it easier to accommodate a larger number of custom error messages. The *Error Handling Programming Guide for Cocoa* gives many examples.

You have two independent error situations to handle. For convenience, you will deal with both of them at the same time. The overall approach is, first, to write two methods to create and return NSError objects configured specially for the particular errors encountered, one method for each error. You call these in the `-openDocumentWithContentsOfURL:display:error:` method to populate the `error` parameter. Second, override NSDocumentController's `-willPresentError:` method, in which you will substitute your new error objects for the default error objects that Cocoa would normally present. Finally, for the second error, you will implement the `-attemptRecoveryFromError:optionIndex:` method declared in the NSErrorRecoveryAttempting informal protocol to give the user an opportunity to correct that error.

Start by writing very simple implementations of these methods. Shortly, you will discard them and write more complex but useful versions.

First, replace the two error statements in the `-openDocumentWithContentsOfURL:display:error:` method so that, instead of returning by indirection the default error provided by NSError's `+errorWithDomain:code:userInfo:` method using the `NSURLErrorCannotOpenFile` error code, you return a new, customized error. One error will be generated by the `-diaryDocumentAlreadyOpenError` method you are about to write, and the other will be generated by the `-incorrectDiaryDocumentErrorForURL:` method you are about to write. The first replacement statement is this:

```
*outError = [self diaryDocumentAlreadyOpenError];
```

The second replacement statement is this:

```
*outError = [self incorrectDiaryDocumentErrorForURL:fileURL];
```

Next, write the first of the two new methods. Both of them construct and return an NSError object with a `userInfo` dictionary using keys defined in NSError. Declare the first at the end of the VRDocumentController.h header file like this:

```
#pragma mark ERROR HANDLING
```

```
- (NSError *)diaryDocumentAlreadyOpenError;
```

Define it in the VRDocumentController.m implementation file like this:

```
#pragma mark ERROR HANDLING
```

```
- (NSError *)diaryDocumentAlreadyOpenError {
    NSString *errorDescription =
        NSLocalizedString(@"The Chef's diary is already open.",
            @"description text for diary document already open error");
```

(code continues on next page)

```

NSMutableDictionary *errorInfo =
    [NSMutableDictionary dictionaryWithObject:errorDescription
    forKey:NSLocalizedStringKey];
return [NSError errorWithDomain:NSURLErrorDomain
    code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}

```

This is the second method. Declare it at the end of the VRDocumentController.h header file like this:

```
- (NSError *)incorrectDiaryDocumentErrorForURL:(NSURL *)fileURL;
```

Define it in the VRDocumentController.m implementation file like this:

```

- (NSError *)incorrectDiaryDocumentErrorForURL:(NSURL *)fileURL {
    NSString *fileName;
    if (floor(NSFoundationVersionNumber)
        <= NSFoundationVersionNumber10_5) {
        fileName = [[fileURL path] lastPathComponent];
    } else {
        fileName = [fileURL lastPathComponent];
    }
    NSString *errorDescription = [NSString stringWithFormat:
        NSLocalizedString(@"The document "%@" is not the current
        Chef's Diary document.", @"description text for incorrect
        diary document error"), fileName];
    NSMutableDictionary *errorInfo =
        [NSMutableDictionary dictionaryWithObject:errorDescription
        forKey:NSLocalizedStringKey];
    return [NSError errorWithDomain:NSURLErrorDomain
        code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}

```

The last statement in both methods is almost identical to the statements you initially inserted in the `-openDocumentWithContentsOfURL:display:error:` method at the beginning of this step. The only difference is that it passes a local `errorInfo` dictionary instead of `nil` in the `userInfo` parameter. The rest of the statements construct the `errorInfo` dictionary, first creating a custom localized `errorDescription` string and then inserting it into the `errorInfo` dictionary.

The second method takes the URL of the document that the user is attempting to open as a parameter, because it will eventually use this URL to give the user an opportunity to make it the new current Chef's Diary document. For the moment, it uses the name only for display in the error description. Its name, including its file extension, is obtained from NSURL's `-lastPathComponent` method, which is new in Snow Leopard, or from the similar NSString method, also named `-lastPathComponent`, if the application is running under Leopard.

3. If you were to build and run the application now and run through an operation that generated one of the errors, you would not see your new error description in the error alert. Instead, you would see the same uninformative error description that `NSDocumentController` presented before you began customizing the error message. This is because Cocoa buried the new description string in an “underlying” dictionary in the error object’s `userInfo` dictionary, keeping the original string at the top level of the error object. Without some help from your code, Cocoa presents only the top-level error description. The Cocoa error-handling system is hierarchical, building layer upon layer of error objects containing information from lower levels. For example, any `NSCocoaErrorDomain` error object might contain an underlying error object from the `NSPOSIXErrorDomain` domain. In order to present your customized error alert, you have to construct it from the materials that `NSDocumentController` and `NSError` provide to you.

To construct and display the new, customized error object, implement `NSDocumentController`’s `-willPresentError:` method. Methods with the same name are declared in `NSDocument` and `NSResponder`, and `NSApplication` declares a delegate method named `-application:willPresentError:` that works in a similar manner. These methods allow you to present an error alert in many different places in an application. In this case, implement `-willPresentError:` in `VRDocumentController` to keep it in the same file where the error was caught. Add this trial implementation at the end of the `VRDocumentController.m` implementation file:

```
- (NSError *)willPresentError:(NSError *)error {
    if ([[error domain] isEqualToString:NSURLErrorDomain]) {
        switch([error code]) {
            case NSURLErrorCannotOpenFile: {
                NSError *underlyingError = [[error userInfo]
                    valueForKey:NSUnderlyingErrorKey];
                if (underlyingError) return underlyingError;
            }
        }
    }
    return [super willPresentError:error];
}
```

The `-willPresentError:` method is called by `NSDocumentController`’s `-openDocument:` action method as soon as the `-openDocumentWithContentsOfURL:display:error:` method encounters the error and returns `nil`. Cocoa passes the call up an error-handling chain that is very similar to the responder chain you encountered earlier, looking for an object that implements the method. When it finds an implementation, it stops looking and calls the method to present the error.

Cocoa passes the error object that you have constructed into the `-willPresentError:` method in the first parameter. The method tests it to make sure that it is an `NSURLErrorDomain` error and, if so, tests its error code. You may want to handle other error codes later, so you use a `switch` statement rather than an `if` statement to test the error code. If it is `NSURLErrorCannotOpenFile`, the method gets down to business.

For now, the method attempts to get the underlying error—the error containing your custom error description string—from the incoming error’s `userInfo` dictionary. If the error object contains an underlying error, the method returns it. Cocoa’s error-handling system does the rest, presenting it to the user. If the error object does not contain an underlying error, or if the error was from any other error domain or had any other error code, the method returns the superclass’s implementation to ensure that any other errors are presented to the user.

4. The trial approach you have just taken is not the simplest way to handle this error. It works, and it is appropriate when you don’t like the wording of `NSDocumentController`’s error description. However, you should backtrack now and do this by the book to learn how it is normally done.

Instead of customizing the default error description, accept it as is but add a separate custom error failure reason. `NSDocumentController` automatically adds any failure reason string that you provide to the default error description string and presents them together as two sentences in boldface in the error alert. The description itself is meant to be only a description of the error (the document couldn’t be opened), without a reason for the failure (the Chef’s Diary is already open, or the document you are trying to open isn’t the correct diary document). To make it easy for you to get at both the combined string and the separate failure reason string, should you ever need them, `NSError` provides the `-localizedDescription` and `-localizedFailureReason` methods. Somewhat confusingly, the `-localizedDescription` method returns the combined sentences, not just the single sentence associated with the `NSLocalizedStringKey` key.

Start by modifying the `-diaryDocumentAlreadyOpenError` method. Delete the existing body of the method and replace it with new statements so that it reads as follows:

```
- (NSError *)diaryDocumentAlreadyOpenError {
    NSString *errorFailureReason = NSLocalizedString(@"The Chef's diary
        is already open.", @"description text for diary document
        already open error");
    NSDictionary *errorInfo =
        [NSDictionary dictionaryWithObjectsAndKeys:
            errorFailureReason, NSLocalizedStringFailureReasonErrorKey, nil];
    return [NSError errorWithDomain:NSURLErrorDomain
        code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}
```

Next, modify the `-incorrectDiaryDocumentErrorForURL:` method. Delete the routines that declare and define the `fileName` local variable, because the default error description string already displays the name of the offending file specified by the incoming file URL. Also delete the statement defining the `errorDescription` local variable. In its place, add a statement defining a new local variable, `errorFailureReason`, as the localizable string “It is not the current Chef’s Diary document.” Finally, delete the statement that creates the `errorInfo` dictionary, and replace it with a version that is more suited to adding additional key-value pairs. Here is the revised method in full:

```
- (NSError *)incorrectDiaryDocumentErrorForURL:(NSURL *)fileURL {
    NSString *errorFailureReason = NSLocalizedString(@"It is not the
        current Chef's Diary document.", @"failure reason text for
        incorrect diary document error");
    NSDictionary *errorInfo =
        [NSDictionary dictionaryWithObjectsAndKeys:
            errorFailureReason, NSLocalizedStringFailureReasonErrorKey, nil];
    return [NSError errorWithDomain:NSURLErrorDomain
        code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}
```

Also revise `-willPresentError:`. Simply delete the two statements accessing the underlying error and instead return the incoming error object as is. Here is the revised method:

```
- (NSError *)willPresentError:(NSError *)error {
    if ([[error domain] isEqualToString:NSURLErrorDomain]) {
        switch([error code]) {
            case NSURLErrorCannotOpenFile: {
                return error;
            }
        }
    }
    return [super willPresentError:error];
}
```

Now build and run the application, open the current Chef’s Diary again, and try to open `Backup.vrdiary`. This time, the error alert reads, “The document ‘Backup.vrdiary’ could not be opened. It is not the current Chef’s Diary.” (**Figure 6.4**). Much better!

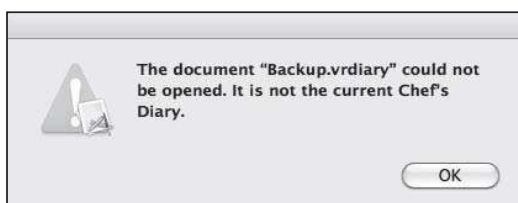


FIGURE 6.4. The revised error alert presented when the user attempts to open a diary document when the current Chef’s Diary document is already open.

The truth is that you no longer need the `-willPresentError:` method at all, because it is now doing nothing more than what any document-based application does automatically. Without your implementation of `-willPresentError:`, the application will automatically present the error alert for you. You need to implement it yourself only when you want to provide some custom error handling that the default error-handling mechanism doesn't provide. There is no harm in leaving the method where it is, however, and you will use it again in a moment to learn some additional techniques.

5. Now that you have it working, add information to the error objects to help the user decide what to do. After that, you will even use facilities provided by `NSError` to perform the recovery action chosen by the user for the second error.

`NSError` declares several keys for use in the `userInfo` dictionary. You have already used three of them, `NSLocalizedStringKey`, `NSLocalizedStringFailureReasonErrorKey`, and `NSUnderlyingErrorKey`. Now you will use four more, `NSLocalizedStringRecoverySuggestionErrorKey`, `NSLocalizedStringRecoveryOptionsErrorKey`, `NSRecoveryAttemptErrorKey`, and `NSURLErrorKey`. `NSError` permits you to include private key-value pairs in the `userInfo` dictionary, and you will do that as well.

Leave the `-openDocumentWithContentsOfURL:display:error:` method as it is.

Change `-diaryDocumentAlreadyOpenError` only slightly, to make the failure reason a little clearer and to add a brief recovery suggestion and an additional key-value pair identifying the error by a unique name. In full, it should now read as follows:

```
- (NSError *)diaryDocumentAlreadyOpenError {
    NSString *errorFailureReason = NSLocalizedString(@"The current
        Chef's Diary is already open.", @"description text for
        diary document already open error");
    NSString *recoverySuggestion = NSLocalizedString(@"Close it and
        try again to make a different document the current Chef's
        Diary.", @"recovery suggestion text for diary document
        already open error");
    NSDictionary *errorInfo = [NSDictionary dictionaryWithObjectsAndKeys:
        errorFailureReason, NSLocalizedStringFailureReasonErrorKey,
        recoverySuggestion, NSLocalizedStringRecoverySuggestionErrorKey,
        ERROR_DIARY_DOCUMENT_ALREADY_OPEN, ERROR_NAME_KEY, nil];
    return [NSError errorWithDomain:NSURLErrorDomain
        code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}
```

The method declares two strings and inserts them into the `userInfo` dictionary with the `NSLocalizedStringFailureReasonErrorKey` and `NSLocalizedStringRecoverySuggestionErrorKey` keys. It also inserts into the dictionary a private name describing the error and a private key, just in case it proves useful for debugging purposes or to distinguish between errors having the same domain and code. You will define these shortly.

The changes to `-incorrectDiaryDocumentErrorForURL:` are more extensive:

```
- (NSError *)incorrectDiaryDocumentErrorForURL:(NSURL *)fileURL {
    NSString *currentDiaryDocumentName = @"";
    NSURL *currentDiaryDocumentURL = [self currentDiaryURL];
    if (currentDiaryDocumentURL != nil) {
        if (floor(NSFoundationVersionNumber)
            <= NSFoundationVersionNumber10_5) {
            currentDiaryDocumentName =
                [[currentDiaryDocumentURL path] lastPathComponent];
        } else {
            currentDiaryDocumentName =
                [currentDiaryDocumentURL lastPathComponent];
        }
    }
    NSString *errorFailureReason = NSLocalizedString(@"It is not the
        current Chef's Diary.", @"failure reason text for incorrect
        diary document error");
    NSString *recoverySuggestion = [NSString stringWithFormat:
        NSLocalizedString(@"Click Open As New Diary to make this
        document the current Chef's Diary and open it. Click Open
        Current Diary to open the current Chef's Diary, "%@.",
        @"recovery suggestion text for incorrect diary document error"),
        currentDiaryDocumentName];
    NSArray *recoveryOptions = [NSArray arrayWithObjects:
        NSLocalizedString(@"Cancel", @"recovery options title for
        Cancel button for incorrect diary document error"),
        NSLocalizedString(@"Open Current Diary", @"recovery options
        title for Open Current Diary button for incorrect diary
        document error"), NSLocalizedString(@"Open As New Diary",
        @"recovery options title for Open As New Diary button for
        incorrect diary document error"), nil];
    id recoveryAttempter = self;
    NSDictionary *errorInfo = [NSDictionary dictionaryWithObjectsAndKeys:
        errorFailureReason, NSLocalizedStringFailureReasonErrorKey,
        recoverySuggestion, NSLocalizedStringRecoverySuggestionErrorKey,
        recoveryOptions, NSLocalizedStringRecoveryOptionsErrorKey,
        recoveryAttempter, NSRecoveryAttempterErrorKey,
        fileURL, NSURLErrorKey,
        ERROR_INCORRECT_DIARY_DOCUMENT, ERROR_NAME_KEY, nil];
    return [NSError errorWithDomain:NSURLErrorDomain
        code:NSURLErrorCannotOpenFile userInfo:errorInfo];
}
```

The method begins by constructing the name of the current diary document from the URL saved in user defaults, so that you can display it in the error alert. You get the URL from user defaults the same way you did in the `-openDiaryDocument:` action method earlier in this recipe, using the `-currentDiaryURL` getter method. You get its last path component—its name and file extension—the same way you extracted the last path component from the `fileURL` parameter a moment ago.

The method next constructs several strings for the error alert, storing each in a local variable. The `errorFailureReason` string is the same as before. New are the `recoverySuggestion` string and the `recoveryOptions` array of strings. The `recoveryOptions` array contains the titles of three buttons that will appear in the error alert, in reverse order. In the alert, the first string, `@Cancel`, will be the default button on the right. The second string, `@Open Current Diary`, will be adjacent to it on its left. The third string, `@Open As New Diary`, will appear some distance to the left, separated from the first two buttons. The `recoverySuggestion` string is closely related to the button titles. It will appear in plain text in the error alert, below the error description. It consists of a couple of sentences explaining to the user the available options.

Cocoa's default error strings use curly quotes, not straight quotes. Here, be careful to use curly quotes yourself for consistency of appearance.

Another local variable, `recoveryAttempter`, is an object of type `id`. In some circumstances, you would find it advantageous to create a separate class or classes to handle recovery from errors. Here, however, it is convenient to designate `self` as the recovery attempter. Whatever object you designate, it must conform to the `NSErrorRecoveryAttempting` informal protocol. You will take care of this shortly by implementing one of the two optional methods declared in that informal protocol.

Finally, the method places the values of all of the local variables into the `userInfo` dictionary with their corresponding keys listed above. It also inserts the URL of the file that the user attempted to open, the `fileURL` argument, with the `NSURLErrorKey` key, another key supplied by `NSError`. And it inserts the private `ERROR_NAME_KEY` key with the value `ERROR_INCORRECT_DIARY_DOCUMENT` to identify this error object.

When the running application calls `-willPresentError:`, after the user attempts to open a diary document that is not the current Chef's Diary, it presents the fully populated error alert containing the error description and reasons, the recovery options, and the three buttons (**Figure 6.5**).

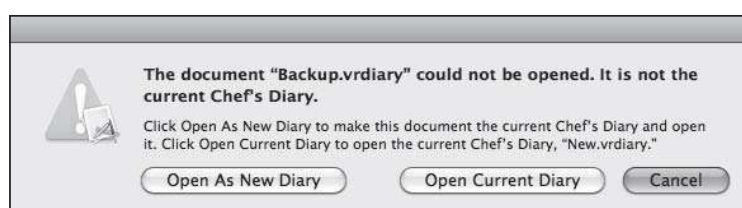


FIGURE 6.5. The final error alert presented when the user attempts to open a backup Chef's Diary document.

To help avoid typographical errors, it is convenient to use defined terms for strings that will be used more than once in the file. Here, you define `ERROR_NAME_KEY`, `ERROR_DIARY_DOCUMENT_ALREADY_OPEN`, and `ERROR_INCORRECT_DIARY_DOCUMENT` to hold the strings in a custom key-value pair that you include in the `userInfo` dictionary. They will not be used outside the `VRDocumentController.m` file, so declare them near the top of the file, just after the existing definition of `DIARY_DOCUMENT_IDENTIFIER`, as follows:

```
#define ERROR_NAME_KEY                @"VRErrorNameKey"
#define ERROR_DIARY_DOCUMENT_ALREADY_OPEN @"diary document already open"
#define ERROR_INCORRECT_DIARY_DOCUMENT  @"incorrect diary document"
```

The key-value pairs you inserted into the `userInfo` dictionary can be used in the `-attemptRecoveryFromError:optionIndex:` method you are about to write, to avoid collisions between errors that have the same error domain and code but different recovery buttons.

6. Revise the `-willPresentError:` method to make use of the new key-value pairs in the `userInfo` dictionary, as shown here:

```
- (NSError *)willPresentError:(NSError *)error {
    if ([[error domain] isEqualToString:NSURLErrorDomain]) {
        switch([error code]) {
            case NSURLErrorCannotOpenFile: {
                if ([[error userInfo] objectForKey:ERROR_NAME_KEY]
                    isEqualToString:ERROR_DIARY_DOCUMENT_ALREADY_OPEN
                    || [[error userInfo] objectForKey:ERROR_NAME_KEY]
                    isEqualToString:ERROR_INCORRECT_DIARY_DOCUMENT]) {
                    return error;
                }
            }
        }
    }
    return [super willPresentError:error];
}
```

The test for the value of the `ERROR_NAME_KEY` is not necessary at this time because you perform no special handling unique to these errors, but it will leave you free to handle other errors differently if you decide to provide additional error handling in the future.

7. There is one more task: implementing the `NSErrorRecoveryAttempting` informal protocol. The documentation makes this sound a little intimidating, but nothing could be easier.

Here is the implementation, at the end of the `VRDocumentController.m` implementation file, in full:

```
- (BOOL)attemptRecoveryFromError:(NSError *)error
    optionIndex:(NSUInteger)recoveryOptionIndex {
    if ([[error domain] isEqualToString:NSURLErrorDomain]) {
        switch([error code]) {
            case NSURLErrorCannotOpenFile: {
                if ([[error userInfo] objectForKey:ERROR_NAME_KEY]
                    isEqualToString:ERROR_INCORRECT_DIARY_DOCUMENT)) {
                    if (recoveryOptionIndex == 1) {
                        [self openDiaryDocument:nil];
                        return YES;
                    } else if (recoveryOptionIndex == 2) {
                        [self setCurrentDiaryURL:[error userInfo]
                                                objectForKey:NSURLErrorKey]];
                        [self openDiaryDocument:nil];
                        return YES;
                    }
                }
            }
        }
    }
    return NO;
}
```

After performing tests on the error domain, the error code, and the private error name similar to those that the `-willPresentError:` method performed, this method provides error recovery actions for two of the buttons in the incorrect diary document error alert. The first, for the Open Current Diary button, simply calls the existing `-openDiaryDocument:` action method, and then returns `YES` to indicate that recovery was successful. That action method has already been written to obtain the URL of the current Chef's Diary document from user defaults and open it. The second, for the Open As New Diary button, does the same thing, except that it first sets user defaults to make the new URL the current Chef's Diary document in user defaults, using the same code you used when writing the `-saveToURL:ofType:forSaveOperation:error:` method in `DiaryDocument.m` earlier in this recipe. If the user clicked the Cancel button, the method does nothing except fall through and return `NO` to indicate that recovery was not successful.

8. Before leaving the topic of error handling, take a look at the `FIXME:` comment you left in the `-openDiaryDocument:` action method in `VRDocumentController.m`. You originally placed the comment here in Step 2 as a reminder to come back

and consider the issue of error handling. The question is whether you should do something other than pass `NULL` in the last parameter when you call the `-makeDocumentWithContentsOfURL:ofType:error:` method.

The same issue arises in several other methods in `VRDocumentController.m` where similar methods are called with `NULL` passed in the `error` parameter. It would be a good idea to flag all the places in your code where you might need to add error handling, in case you want to revisit the issue later. Add the same `FIXME:` comment to the `-newDiaryDocument:`, `-aliasDataFromURL:`, and `URLFromAliasData:` methods in `VRDocumentController.m`.

Upon reflection, the answer for now might be that there is no need to change most of these methods. There is no need to add error handling. It is perfectly acceptable to pass `NULL` to Cocoa methods that have a pass-by-reference `error` parameter, if you have no special reason to expect an unusual or application-specific problem.

Whether or not you pass `NULL` in these `error` parameters, you always have to be careful to test the return value of a method containing a pass-by-reference `error` parameter to see whether it is `nil` or `NO`. Such methods are required to return `nil` or `NO` if there was an error, and in such a case the `error` parameter is guaranteed to contain an `NSError` object for your use, should you choose to take advantage of it. Even if you don't use the error object, you're bound to get into trouble if you try to use a `nil` return value.

As to the original method where you inserted a `FIXME:` comment, however, `-openDiaryDocument:`, the best answer is yes, you should handle errors here. The reason for this is that you call `-openDiaryDocument:` yourself in the new `-attemptRecoveryFromError:optionIndex:` method that you just added. In both cases, the new method blithely assumes that the call succeeds in opening the document, returning `YES` accordingly. But what if the call fails? The user's attempt at recovery would fail silently, a black mark against Vermont Recipes.

To test how the application responds to an error here, temporarily change `-openDiaryDocument:` so as to pass `[NSURL fileURLWithPath:@"/nosuchfile"]` in the first parameter of the call to `-makeDocumentWithContentsOfURL:ofType:error:`. Build and run the application, and double-click `Backup.vrdiary` in the Finder. In the resulting error alert, click the Open Current Diary button. Nothing happens. The same is true if you simply choose `File > Open Chef's Diary`.

Now change `-openDiaryDocument:` again. This time, declare a local variable named `error` by inserting the following statement immediately above the call to `-makeDocumentWithContentsOfURL:ofType:error:`, like this:

```
NSError *error;
```

Then change the `NULL` argument in the `error` parameter of `-makeDocumentWithContentsOfURL:ofType:error:` to `&error`. Finally, substitute this block for the `if (diary != nil)` statement immediately following the method:

```
if (diary == nil) {  
    [self presentError:error];  
} else {
```

You should also remove the `FIXME:` comment, since you've now fixed it.

Build and run the application again, double-click `Backup.vrdiary`, choose `Open Current Diary`, and you see a meaningful error alert (**Figure 6.6**). The same thing happens if you choose `File > Open Chef's Diary`.

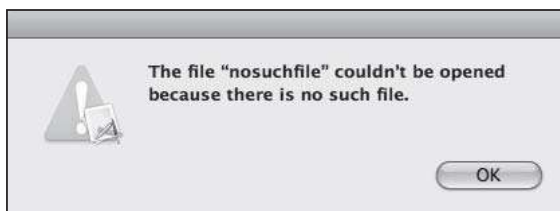


FIGURE 6.6. The error alert presented when the user tries to open a URL representing a nonexistent document.

The `-presentError:` call sent the error up the error-handling chain, looking for an implementation of `-willPresentError:`. It found your implementation in `VRDocumentController.m`, but your implementation doesn't catch errors in the `NSCocoaErrorDomain` error domain with error code 260, which is what this is. As a result, your implementation fell through to call the superclass's implementation, which presented the alert. It would have worked the same if you had not implemented `-willPresentError:` at all.

Be sure to change the `[NSURL URLWithString:@"nosuchfile"]` temporary argument back to `fileURL`, and you're done. For the record, here is the final version of `-openDiaryDocument:`.

```
- (IBAction)openDiaryDocument:(id)sender {  
    NSDocument *diaryDocument = [self diaryDocument];  
    if (diaryDocument != nil) {  
        [[[[diaryDocument windowControllers] objectAtIndex:0] window]  
         makeKeyAndOrderFront:sender];  
    } else {  
        NSURL *fileURL = [self currentDiaryURL];  
        if (fileURL != nil) {  
            NSError *error;  
            DiaryDocument *diary =  
                [self makeDocumentWithContentsOfURL:fileURL  
                 ofType:DIARY_DOCUMENT_IDENTIFIER error:&error];  
            if (diary == nil) {  
                [self presentError:error];  
            }  
        }  
    }  
}
```



```

        } else {
            [self addDocument:diary];
            [diary makeWindowControllers];
            [diary showWindows];
        }
    }
}
}

```

9. Save a snapshot. Name it *Recipe 6 Step 3*, and add a comment saying, **Implemented error handling for diary document.**

Step 4: Prepare Localizable Strings for Internationalization

In Step 11 of Recipe 1, you learned that your application should include special files known as *strings files*, each containing keys and values for localizable strings. These allow your application to be translated into other languages without your having to revise the code. In this recipe, you have called the `NSLocalizedString()` macro many times, and you called it a couple of times in Recipe 4 as well. Now you need to take care of this important housekeeping matter.

Rather than going to the trouble of finding all those macro calls yourself and typing them into the `Localizable.strings` file you created in Recipe 1, add a script build phase to the project now to perform the task automatically. After all, that's what computers are for.

In a major project, you would normally run the *genstrings* command-line tool to do this when you are preparing your application for deployment so as to avoid time lost running it every time you build your project. However, I have found that projects the size of Vermont Recipes build very quickly anyway, so I am in the habit of placing a script build phase in my application target that runs *genstrings* every build. That's what you will do now.

1. In the main Vermont Recipes project window, expand the Targets group in the Groups & Files pane and then expand the Vermont Recipes target so that you can watch the action. You see several build phases in the target.
2. Select the Vermont Recipes target itself and choose Project > New Build Phase > New Run Script Build Phase. A new Run Script subgroup appears at the end of the expanded list of build phases. In addition, a window opens with several

fields into which you can type text. The field at the top, Shell, already contains the text `/bin/sh`.

3. In the field labeled Script, type the following text:

```
echo Generate Localizable.strings from source and install in
    English.lproj folder of built product

genstrings -o "${TARGET_BUILD_DIR}/${PRODUCT_NAME}.${WRAPPER_EXTENSION}"/
    Contents/Resources/English.lproj" *m

exit 0
```

This text contains three commands, `echo`, `genstrings`, and `exit`. The first two appear wrapped as you see them here, but you should enter them as single (one-line) statements.

When you build the project, the text following the `echo` command appears in the Xcode Build Results window if you have chosen All Messages in the pop-up menu under the search field. This makes it easy to see that this script build phase has run when you build the project.

The `genstrings` command takes all implementation files in the project folder, scans them for `NSLocalizedString()` and similar macros, and constructs a properly formatted UTF-16 strings file in the built product package's Resources folder.

The Localizable.strings file that you put in the project in Recipe 1 isn't needed, because you aren't filling it out manually, but there is no harm leaving it in place. The script works directly on the built product.

4. Build the project.
5. To examine the result of the script build phase, go to your project's build folder and open it. You should find the build folder in the project folder. Open the Release or Debug subfolder, depending on what you built, and Command-click (or right-click) the Vermont Recipes application icon. Choose Show Package Contents and open the Contents, Resources, and English.lproj folders in turn. You should find a Localizable.strings file there, along with other localizable files you expect to see in the English.lproj folder, such as the Read Me file. Open the Localizable.strings file in Property List Editor or in a text editor such as TextEdit. What you see is a key-value pair for every `NSLocalizedString()` or similar macro in your implementation files, each headed by the comment you provided.

Step 5: Build and Run the Application

You have now implemented a robust and user-friendly way to let the user maintain a single Chef's Diary document with any name and in any location the user desires. At the same time, if the user does create a second version of the Chef's Diary, whether by choosing File > Save As in Vermont Recipes or by making a duplicate in the Finder, the user can easily switch back by choosing File > Open or File > Open Recent in Vermont Recipes, or by double-clicking an old version in the Finder. The error handling you have added protects the user against mistakes while at the same time offering recovery options giving the user maximum flexibility.

To try out the finished mechanism, build and run Vermont Recipes. Assuming that the `New.vrdiary` file is still the current Chef's Diary document, choose File > Open in Vermont Recipes, select `Backup.vrdiary`, and click Open. Alternatively, choose File > Open Recent and choose `Backup.vrdiary` from the submenu, or double-click `Backup.vrdiary` in the Finder. The error dialog opens, warning you that `Backup.vrdiary` can't be opened because it is not the current Chef's Diary. Click the Open Current Diary button, and `New.vrdiary` immediately opens.

Repeat the process, but click the Open As New Diary button, and `Backup.vrdiary` immediately opens. Either way, the file you opened is now the current Chef's Diary.

Step 6: Save and Archive the Project

Quit the running application, close the Xcode project window, and save if asked. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 6.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 7.

Conclusion

In Recipe 6, you focused on techniques to implement a one-of-a-kind document in a document-based application. In the process, you learned how to save values behind the scenes in user defaults and retrieve them, how to create and resolve alias records and bookmarks, and how to use the powerful and flexible error-handling mechanism built into Cocoa's document-handling facilities. You also learned several ways to keep your code well organized and how to prepare localized strings for internationalization.

In the next recipe, you will refine the application's GUI by tidying up a few issues regarding the application's windows and by implementing a number of other features that are common in Mac applications.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 6.

General Documentation

Cocoa Fundamentals Guide

Xcode Workspace Guide

Application Architecture Overview

Binary Data Programming Guide for Cocoa

Alias Manager Reference

Carbon-Cocoa Integration Guide (Using the FSRef Data Type)

Low-Level File Management Programming Topics (Resolving Aliases)

Storing File References in CFPreferences, Technical Q&A QA1350

CFURL Reference

NSURL Class Reference

NSURL and CFURL Release Notes for Mac OS X 10.6

Folder Manager Reference (DeterminelfPathsEnclosedByFolder)

Cocoa Scripting Guide (How Cocoa Applications Handle Apple Events)

Error Handling Programming Guide for Cocoa

Document-Based Applications Overview (Error Handling in the Document Architecture)

Internationalization Programming Topics (Localizing String Resources)

Resource Programming Guide (String Resources)

genstrings man page

Refine the Document's Usability

You have now created the beginnings of a working application. In this book, you will not complete the recipes document and the Core Data database that manages its content, but the Chef's Diary window and its underlying document are fully functional if not yet fully configured.

Pretend for the time being that the application is to consist of nothing more than the Chef's Diary. In this recipe and the rest of the recipes in Section 2, turn to details that are best left until core functionality has been implemented. Since the Chef's Diary has reached that stage, you are ready to undertake tasks that you should complete before releasing any application to its intended market. It's time for the spit and polish that make a good application into an outstanding application.

In this recipe, you refine the diary document by cleaning up all sorts of little problems that make it less than perfect, mostly in its graphical user interface. For example, you set the default size and placement of the diary window; you autosave its position on the screen when the user closes it so that it will reopen in the same place on the screen; you set its default zoom size; and you make its Revert to Saved menu item work. While you're at it, you improve some aspects of the recipes window's behavior.

In subsequent recipes in Section 2, you will add other features that are required or highly recommended for any finished application. You will add features that improve the performance of the application and the system as a whole, such as

HIGHLIGHTS

Setting the minimum and maximum sizes of a document window

Creating categories that add methods to existing Cocoa classes

Setting the initial position and size of a document window

Setting the Standard Zoom Size of a document window

Autosaving document window configurations

Using Cocoa notifications

Autosaving document contents

Restoring autosaved documents

Customizing the restoration process

Presenting an informational alert to the user

Allowing the user to suppress future informational alerts

Backing up a document

Reverting a document to its last saved contents

support for Snow Leopard’s new sudden termination technology. And you will take another look at the project’s build settings. Among other things, it’s about time you made sure it works when running under Leopard as well as Snow Leopard, and on PowerPC Macs as well as Intel Macs. In addition, you will implement printing support, user preferences, a Help book, and AppleScript support, and you will prepare it for the application’s deployment to its intended audience.

Step 1: Set the Minimum and Maximum Sizes of the Document Windows

In this and the next few steps, you will continue to clean up the diary document by setting its window’s minimum and maximum sizes, by setting its window’s initial size and position, by autosaving its window’s size and position and the position of the split view’s divider so that they are automatically restored when the window is closed and reopened, and by taking other steps to ensure that its window behaves in accordance with Apple’s guidelines.

Start with the minimum and maximum sizes of the diary document’s window, and then consider the recipes window as well.

1. Leave the archived Recipe 6 project folder where it is, and open the working Vermont Recipes subfolder.

Increment the Version in the Properties pane of the Vermont Recipes target’s information window from 6 to 7 so that the application’s version is displayed in the About window as 2.0.0 (7).

2. Open the DiaryWindow nib file and select its Window object. You set the minimum and maximum sizes of the window in the Window Size inspector. There are no hard and fast rules, but it makes sense for a text document like the Chef’s Diary to be no wider than an easily scanned line of text. The onscreen diary window is not paginated, and Apple suggests that you size a document vertically to expose as much of its content as possible. I think of a diary as usually being smaller than a sheet of typewriter paper—say, with a typical size of about 6 inches by 9 inches. It might be as large as a sheet of typewriter paper, 8.5 by 11 inches in the United States. Or it might be as small as, say, 4 by 6 inches. In no event should you let it be sized so narrow that the controls at the bottom of the window overlap.

You can set the minimum and maximum sizes visually by resizing the window design surface in Interface Builder to the desired size and clicking a setting in the Window Size Inspector. Remember to hold down the Command key while resizing it so that all of its subviews resize at the same time. You can also hold down the Shift key to constrain resizing to the horizontal, vertical, or diagonal direction.

First, Command-drag the resize control to set the window to its desired minimum size, and then click Use Current in the Minimize Size section. Then resize the window to its desired maximum size and click Use Current in the Maximum Size section. In both cases, the corresponding Width and Height text fields update to show the new minimum and maximum sizes, and the checkbox is selected for each to show that it is now in effect.

I prefer to set the sizes by typing round numbers into the Width and Height text fields in the Window Size Inspector. If you do it this way, enter **400** by **550** pixels for the minimum size in the Width and Height text fields, respectively, and enter **850** by **1100** pixels for the maximum size. Press Enter, or click or tab out of each text field, to commit the new value.

3. Save the nib file, and build and run the application. Choose File > New Chef's Diary. Never mind where on the screen the window opens or what its initial size is—you'll set those up in the next step.

Now drag the window's resize control, and you see that you cannot make it larger than the maximum size or smaller than the minimum size you just set. This remains true even if you reposition the window on the main screen or move it to a secondary screen.

4. You might as well set the minimum size of the recipes window while you're at it.

Like the iTunes and Mail windows, which have a similar appearance, the recipes window's maximum size should be constrained only by the size and shape of the available screen real estate. You can't do that in Interface Builder, so you'll do it in code in a moment.

You can, however, set a reasonable minimum size for the recipes window using Interface Builder. The recipes window should be wider than it is tall. To my taste, Mail allows you to set its main window too small, but iTunes seems about right.

Open the RecipesWindow nib file, and then open its Window object and select the Window Size inspector. Command-drag the Window's design surface until it is about 700 pixels wide by 350 pixels high, or enter those numbers in the Minimum Size section of the inspector and choose Use Current.

5. Save the nib file and run the application again. The recipes window opens automatically. Use its resize control and eyeball it to verify that you can't make it smaller than about 700 by 350 pixels.

6. Finally, write a little code to set the maximum size of the recipes window. This takes several steps.

Before proceeding, consider another detail for the application's specification. The user may be able to get the best use out of the recipes window when it is about as big as the display, but its usability will be degraded if it is so big that it stretches across two displays. You shouldn't be overly intrusive and stop a determined user from dragging the window to a position straddling the boundary between two displays. But you can prevent the user from resizing it so that it is bigger than the largest display available. This is what iTunes does, for example; if you make its window as large as the largest display and then drag it partway onto an adjacent display, you still can't make it any wider. Among other things, this ensures that the user is always able to place the window on a display where both its title bar and its resize control are reachable. Many applications aren't very careful about dealing with multiple displays, but you should try to take into account users who have two or more of them.

You get the size of the largest display currently available by using Cocoa's `NSScreen` class. Define *largest* for these purposes as total visible area, as a rough measure of information capacity. You could define it differently—for example, by using the maximum width of one display and the maximum height of another display—but your goal for this window is to be able to fill the available space on one display, and you don't want to get hung up on trying to decide whether that means the widest or the tallest display. Note that, if the user has two displays of identical dimensions, the larger one as defined here will be the one without the menu bar and the Dock, because the `NSRect` returned by the `-visibleFrame` method that you will use in a moment excludes the menu bar and the Dock.

Getting the largest screen is a discrete task, so it makes sense to write a separate method to do it. Here, write a method named `+VR_screenWithLargestVisibleFrame`. I usually find it tempting, and all too easy, to write a method like this in the class I'm currently working on, in this case the `RecipesWindowController` class. However, this task is not specific to the recipes window, so it would be much more appropriate to place it in a more general location. An ideal solution, commonly used, is to make it a category method.

You learned about categories in Recipe 6. There, you discussed them primarily as a technique for organizing source files, but they can be used for other purposes. Here, you create a category on an existing Cocoa class, `NSScreen`. Since this is to be a generally available method, you should create the category in a separate file of its own and declare the method as a class method.

In Xcode, choose `File > New File`, and create Objective-C class header and implementation files under the name `NSScreen+VRScreenAdditions`. This uses

a common naming convention for category files that add methods to an existing class, in which you combine the class name (NSScreen) and the category name (VRScreenAdditions) with a plus sign. The prefix *VR*—for Vermont Recipes—in the filename is not used here to avoid namespace collisions in the usual sense, but only to make it possible to use somebody else’s file creating additional methods for NSScreen, as well as your own. You wouldn’t be able to import two header files having the same name. Apple specifically recommends that you name categories that add methods to an existing class by appending *Additions* to the class name, in *Coding Guidelines for Cocoa*. This and the conventional use of the class name and a plus sign in the filename makes it relatively likely that you might encounter identically named category files in the Cocoa user community if you did not use a unique prefix.

Create a new group in the existing Classes group in the Groups & Files pane of the project window, name it **Categories**, and move the two new source files into it. Since you will call the `+VR_screenWithLargestVisibleFrame` class method in the RecipesWindowController class, add this line now near the top of the RecipesWindowController.m implementation file:

```
#import "NSScreen+VRScreenAdditions.h"
```

Add the usual identifying text to the top of the NSScreen+VRScreenAdditions.h header file, and import `<Cocoa/Cocoa.h>`. Then write the code like this:

```
@interface NSScreen (VRScreenAdditions)
+ (NSScreen *)VR_screenWithLargestVisibleFrame;
@end
```

The category declaration looks just like a class declaration, except that the name of the class in the `@interface` directive is followed by the category name in parentheses, and there are no braces to hold instance variables because you cannot declare new instance variables in a category.

You put the prefix *VR* and an underscore character on the method name, making it `+VR_screenWithLargestVisibleFrame`. In this case, you *are* dealing with a conventional namespace collision issue. If Apple were ever to add a method to NSScreen called `+screenWithLargestVisibleFrame` and you hadn’t put the prefix on your method name, your method and Apple’s new method would clash. In *Coding Guidelines for Cocoa*, Apple recommends the use of a two- or three-letter uppercase prefix with an underscore character for your private methods that might clash with Cocoa private methods, and the convention is extended here to category methods.

The new NSScreen+VRScreenAdditions.m implementation should look like this:

```
#import "NSScreen+VRScreenAdditions.h"

@implementation NSScreen (VRScreenAdditions)

+ (NSScreen *)VR_screenWithLargestVisibleFrame {
    NSArray *screenArray = [NSScreen screens];
    if ([screenArray count] == 1) return [screenArray objectAtIndex:0];

    NSScreen *screen = nil;
    CGFloat largestScreenArea = 0.0;
    CGFloat thisScreenArea = 0.0;
    for (NSScreen *thisScreen in screenArray) {
        thisScreenArea = NSWidth([thisScreen visibleFrame])
            * NSHeight([thisScreen visibleFrame]);
        if (thisScreenArea > largestScreenArea) {
            screen = thisScreen;
            largestScreenArea = thisScreenArea;
        }
    }
    return screen;
}

@end
```

On a system with multiple displays, the `+VR_screenWithLargestVisibleFrame` method iterates over the array of screen objects for all displays currently attached to the computer, using NSScreen's `+screens` class method. It calculates the visible area of each and returns the largest. NSScreen's `-visibleFrame` method excludes the area occupied by the menu bar and the Dock, if the screen currently being tested contains the menu bar and the Dock. The visible frame is an NSRect, a C structure containing **origin** and **size** structures of type NSPoint and NSSize, respectively. You use the convenient NSWidth and NSHeight functions to extract the width and height of the visible frame.

Most users have only one display, so the method first checks whether it is running on such a system and, if so, simply returns that display's screen. It does this by getting the screen at index 0 from the array, which, even in a multiple-display system, is guaranteed to be the screen with the menu bar and Dock. You could just as well have returned the screen obtained with NSScreen's `+mainScreen` class method, since there is only one display available, but you should be aware that `+mainScreen` returns the screen where the window currently having keyboard focus is located, which is not necessarily the screen with the menu bar and the Dock on a multi-display system.

7. Another consideration is that the recipes window has a drawer. An `NSDrawer` object does a pretty good job of dealing with its parent window's position on the screen. By default, the drawer opens on the right side of the window, but if the window is moved too close to the right edge of the display, the drawer automatically opens on the left side of the window. It continues to open on the left until you drag the window too close to the left edge of the display. In fact, it switches back to opening on the right edge of the window, if there's room, even when you move it near the left edge of a display that has a second display located to the left of the main display. All this behavior is built in, including the drawer's preference to open on the same display where its window is located even if an adjacent display is available.

What isn't covered by built-in `NSDrawer` behavior is the case in which the parent window fills the entire display. In that situation, the *Apple Human Interface Guidelines* (HIG) specify that the drawer should open off the screen. It isn't very useful there, however. The *Apple Human Interface Guidelines* grant you a great deal of discretion when it comes to setting the maximum size of a window, and there is no reason why you can't choose to steal enough space from the recipes window to make room for its drawer. Thus, another role for the maximum window size setting is to ensure that the window is never so big that the open drawer can't be seen when it's opened.

Before you can know how much space the drawer needs on the screen, you have to set its minimum and maximum content sizes. You didn't take care of this detail in Recipe 2 because you haven't yet provided any content for the drawer. For the moment, therefore, set arbitrary minimum and maximum widths. To do this, select the drawer object in the `RecipesWindow.nib` file's document window, go to the Drawer Size inspector, and enter **100** in the Min Width constraint and **300** in the Max Width constraint. Now the drawer cannot be opened farther than required to show a content view 300 pixels wide, and when it is dragged toward the closed position, it snaps shut as soon as its content view reaches 100 pixels.

These values relate to the width of the drawer's content view, not of the entire drawer including its border. What you care about for calculating the screen space needed for the drawer is the width of the entire drawer. Every drawer is contained in a window of its own (not to be confused with the drawer's parent window). Although the window containing the drawer is not available through a method of `NSDrawer`, it is readily available through the `-window` method of the drawer's content view, since the drawer's content view is an `NSView` object. The `-minSize` and `-maxSize` methods of the containing window return values calculated by Cocoa from the minimum and maximum sizes of the drawer's content view. Note that the window containing the drawer will never be larger than the drawer's parent window because the drawer, when it is closed, hides behind the parent window, but it can be smaller.

Write a method named `-VR_drawerMaxWidth` to get the maximum width of the entire drawer, including its border, by getting the maximum width of the drawer's containing window. Like the `+VR_screenWithLargestVisibleFrame` method, this is a general-purpose method best implemented in a category, this time a category on `NSDrawer`. It should be an instance method, not a class method, because it will be called on a specific drawer. Create another pair of source files for the new category, naming it `NSDrawer+VRDrawerAdditions`, and place them in the new Categories group in the project window. Near the top of the `RecipesWindowController.m` implementation file, add this statement to import the new header file:

```
#import "NSDrawer+VRDrawerAdditions.h"
```

The `NSDrawer+VRDrawerAdditions.h` header file should look like this:

```
#import <Cocoa/Cocoa.h>

@interface NSDrawer (VRDrawerAdditions)
- (CGFloat)VR_drawerMaxWidth;
@end
```

The implementation file should look like this:

```
#import "NSDrawer+VRDrawerAdditions.h"

@implementation NSDrawer (VRDrawerAdditions)

- (CGFloat)VR_drawerMaxWidth {
    NSWindow *drawerWindow = [[self contentView] window];
    if (drawerWindow && [drawerWindow isKindOfClass:[NSWindow class]])
        return [drawerWindow maxSize].width;
    return 0.0;
}

@end
```

Although getting the drawer's containing window in this fashion is very common, a purist might worry that it depends on a private implementation detail of drawers that Apple could change in a future version of the Cocoa frameworks. Currently, the drawer's containing window is an `NSDrawerWindow` object, but `NSDrawerWindow` is a private Cocoa class neither documented nor visible in `/System/Library/Frameworks`. It plainly inherits from `NSWindow`, however, and it is accessible, not by trickery, but by using published Cocoa methods as described above. It is therefore legal to do this. It is nevertheless prudent to check whether the drawer has a containing object (that is, `drawerWindow` is not `nil`) to ensure that the attempt to return the struct representing its maximum size does not fail, and you throw in a test whether it is a subclass of `NSWindow`. You do this in the

second statement in the method. If in a future version of Mac OS X Apple stops using a containing window, your `-drawerMaxWidth` method will return 0.0, and the window will behave strictly in accordance with *Apple Human Interface Guidelines*: The drawer will open off the screen if the recipes window fills the screen.

This method uses the standard technique for testing whether an object is of a particular class or descended from that class: `[[drawerWindow isKindOfClass:[NSWindow class]]`, using the NSObject protocol's `-class` method available in almost every class in the Cocoa frameworks.

8. A final consideration is that the recipes window has a toolbar. The window usually grows taller when the toolbar is made visible, so you might think that you have to leave room on the screen to make a hidden toolbar visible as you just did with the drawer. In fact, you don't. A window honors its maximum size constraint when the toolbar is toggled to its visible state. If the window is already as tall as it is allowed to get, making the toolbar visible automatically decreases the height of the window's content view instead of increasing the height of the window. Apple's documentation does not disclose this fact, nor does it reveal that `NSWindow`'s `-maxSize` method automatically adjusts its return value to include the height of the toolbar when it is visible.
9. With the supporting category methods under your belt, add these statements at the end of the existing `-windowDidLoad` method:

```
CGRect maxVisibleFrame =  
    [[NSScreen VR_screenWithLargestVisibleFrame] visibleFrame];  
CGFloat drawerMaxWidth =  
    [[[[self window] drawers] objectAtIndex:0] VR_drawerMaxWidth];  
[[self window] setMaxSize:  
    CGSizeMake(maxVisibleFrame.size.width - drawerMaxWidth,  
    maxVisibleFrame.size.height)];
```

You called the new category methods exactly as if they were methods declared in the Cocoa frameworks, in `NSScreen` and `NSDrawer`. To all intents and purposes, they are.

Step 2: Set the Initial Position and Size of the Document Windows

Read the “Positioning Windows,” “Moving Windows,” and “Resizing and Zooming Windows” subsections of the “Windows” section of the *Apple Human Interface Guidelines* for the general principles you should follow in setting the initial position and size of a document window. In summary, the first time a new document window

opens, it should be centered horizontally, its top should butt up against the menu bar or any application toolbar positioned below the menu bar, and it should be tall enough to show as much of its content as possible, given the size of the display. It should not overlap the Dock. If the document's contents are too large to be shown in a single window of reasonable size, you need not have it stretch all the way from the menu bar down to the Dock, especially on a large display. Show the largest sensible unit in the document, such as a page.

These requirements for document windows differ from the requirements for non-document windows, such as dialogs.

1. To set the diary window's initial size, open the `DiaryWindow.nib` file, select the Window, and open its design surface. Command-drag its resize control to set the desired initial size by eye, and then click **Use Current** in the Content Frame section of the Window Size inspector. For the Chef's Diary, make it the typical size of a diary as discussed at the beginning of Step 1, about 6 by 9 inches. To do this by the numbers, enter **600** and **900** in the Width and Height text fields, respectively, at the bottom of the Content Frame section of the inspector.

This initial size is known as the document window's *standard state*.

2. A good way to position the window in the center of the screen when it initially opens is to use the Initial Position section of the Window Size inspector. Drag the little image of a document around on the little image of the screen until the window appears to be centered horizontally and its top is against the menu bar. To get a more precise result, do it by the numbers. Use the Displays pane of System Preferences to get the width of your primary display, and use the figures in the inspector for the x-coordinate and the width to position the window in the exact center of the screen horizontally, although this requires doing a little simple arithmetic. Click **Preview** to move your real window to the final position on the real screen, and decide whether you're happy with it.

It is ordinarily best to leave the two window anchors that are connected to the top and left edges of the window extended to the edges of the screen in the inspector, because Interface Builder then calculates the position of the upper-left corner of the window so that it comes out the same as shown in the inspector, even when you run the application on computers having displays of differing sizes. If you retracted these two anchors, the window's position would be calculated based on its origin in the lower-left corner. To extend or retract one of the anchors, click it. Leave both side anchors and the top anchor extended to ensure that the window is centered horizontally and butted up against the menu bar on any computer system.

If you decide to write some code for custom behavior, it is best placed in the window controller's `-windowDidLoad` method or its `-awakeFromNib` method. Keep

in mind a couple of points. First, don't use `NSWindow`'s `-center` method. It looks inviting, but it places windows "somewhat above center vertically," according to the *NSWindow Class Reference*. This is the initial placement requirement for nondocument windows like dialogs, not for document windows. Also, make sure the Visible At Launch checkbox in the Window Attributes inspector in Interface Builder is deselected. When this setting is selected, it causes the window to be shown as soon as it is loaded from the nib file. While that is convenient for simple, one-window applications, it can cause flashing in some circumstances when a window is displayed under the control of a window controller, as the Vermont Recipes windows are.

3. Go through the same procedure with the recipes window. The initial size should be intermediate between the minimum and maximum sizes, erring on the large side. It should be big enough to show most of the content you expect to appear in a recipes window—say, 1200 by 800 pixels. Be sure to hold down the Command key while you resize it with its resize control. The source list pane on the left may look too wide, but ignore that for now.
4. Save both nib files, and build and run the application on your main computer. The recipes window appears at the intermediate size you just set, centered horizontally on the screen and butted up against the menu bar. Choose File > New Chef's Diary. The diary window opens at its initial size, about 6 by 9 inches, centered horizontally and butted up against the menu bar. Move both windows somewhere else on the screen and resize them. Then close both of them and reopen them. Since you didn't save either document when you closed the windows, these are new documents. Their windows therefore once again appear centered on the screen at their initial sizes.

Step 3: Set the Standard Zoom Size of the Document Windows

The *Apple Human Interface Guidelines* describe zooming a window as toggling it between its *user state* and its *standard state*. The *HIG* defines the standard state of a new document window as its initial size and position, which you set in Step 2. An application can change the standard state programmatically in response to changed conditions—for example, to match the size of a printed page as set from time to time by the user in the Page Setup dialog—but the user can't change the standard state directly. By contrast, the user establishes a new user state for a window every time the user changes its size or position by at least 7 pixels. Clicking the zoom button toggles the window between the two states, changing its size while anchoring the top-left corner in place.

The *HIG* describes what an application should do when the user clicks the zoom button while the window is in the user state. It should of course change to its standard state, but there are a number of qualifications. Among other things, the *HIG* cautions that the window should not normally zoom to fill the entire screen, unless the user has deliberately put it into that state. In addition, the application should move the window so that it is entirely onscreen if zooming it to the standard state would otherwise leave it partially offscreen or straddling two displays. The *HIG* cautions that zooming to the standard state should move the window onto the screen where the bulk of it is already located, and it should not overlap the Dock.

You have already set both windows' initial or standard states and their maximum sizes, and Cocoa automatically takes care of moving the zoomed window fully onto the correct screen, avoiding the Dock. However, you must take additional steps to achieve full compliance with the *HIG*.

The most important deficiency is that, when the diary window or the recipes window is first opened, clicking the zoom button causes the window to expand to its maximum size. It would have caused the window to expand to fill the screen if you had not already set its maximum size smaller than that. The window expands to the maximum size even if you resized it to something other than its standard state before clicking the zoom button. In effect, a new window thinks it is in the user state, and it thinks the standard state is to fill the screen within the limits of its maximum size.

To get the window to behave as prescribed by the *HIG*, implement the `-windowWillUseStandardFrame:defaultFrame:` delegate method. For the Chef's Diary window, implement it in `DiaryWindowController`. Implement it for the recipes window in `RecipesWindowController`. You already designated their window controllers as these windows' delegates.

It is your job, in the delegate method, to return an `NSRect` specifying the desired frame of the zoomed window in its standard state, and also to anchor the window's top-left corner in place to meet users' expectation that resizing always occurs down and to the right. You should start by taking the window passed in the first parameter and getting its frame. In using the frame to devise a new frame to return, you can, if you wish, take account of the second parameter, which contains the visible frame of the window's current screen—that is to say, the screen holding the bulk of the window. The protocol reference erroneously describes this parameter as providing the “size” of the current screen; it actually provides an `NSRect` specifying its origin as well as its size. The origin is specified in global coordinates taking into account multiple displays, if present. Thus, the origin may contain negative values if the current screen is below or to the left of the screen containing the menu bar and the Dock. The parameter holds the visible frame of the screen, not its full frame, omitting the menu bar and the Dock if they are on this screen.

You should return an `NSRect` having whatever size you determine is the window's standard size and an origin adjusted to anchor the top-left corner. The `-zoom:` method will alter the origin you return, if necessary, moving the window fully onto

the current screen. In the process, it may resize the window—for example, if its standard height is taller than the available space on the current screen.

You originally set the window's standard state in the nib file using Interface Builder, but you haven't set up any way to get the standard state `NSRect` into the application and save it. It would be nice if you could capture the window's standard state from the nib file, before the user has had an opportunity to resize the window, and then keep a record of the standard state around for the life of the application. An easy way to do this is to capture the window's initial frame in the window controller's `-windowDidLoad` method, right after the window is loaded from the nib file, and then save the size of the frame in the application's user defaults. Since there is only one Chef's Diary window and one recipes window, you can simply provide a unique user defaults key for each. If, while the application is running, you decide to change the window's standard size, simply write a new value to the user defaults.

1. Start with the diary window. You learned in Recipe 6 how to declare and define a globally available key for use with the user defaults. This key normally wouldn't need to be globally available, but you might want to allow the user to change the standard state of the window when you get around to creating a Preferences window in Recipe 10.

At the top of the `DiaryWindowController.h` header file, just before the `@interface` directive, enter this declaration:

```
extern NSString *VRDefaultDiaryWindowStandardSizeKey;
```

At the bottom of the `DiaryWindowController.m` implementation file, after the `@end` directive, define it like this:

```
NSString *VRDefaultDiaryWindowStandardSizeKey =  
    @"diary window standard size";
```

2. Next, in the `-windowDidLoad` delegate method that you wrote in the `DiaryWindowController.m` implementation file in Recipe 3, add code to save the newly opened window's size to the user defaults, at the end. Here is one way to do it:

```
NSSize standardSize = [[self window] frame].size;  
NSData *standardSizeData =  
    [NSData dataWithBytes:&standardSize length:sizeof(NSSize)];  
[[NSUserDefaults standardUserDefaults] setObject:standardSizeData  
    forKey:VRDefaultDiaryWindowStandardSizeKey];
```

The user defaults can only hold objects, not C structures like `NSSize`, and they can only hold certain types of objects. It is easy to convert an `NSSize` structure to an `NSValue` object, but `NSValue` is not one of the types of object that can be saved in the user defaults. Instead, in this code, you copy the bytes in the `NSSize` structure to an `NSData` object, which can then be saved in the user defaults. This is commonly done using `NSData`'s `+dataWithBytes:length:` class method, as you see here.

While this was a common technique in the days of 32-bit applications that could run only on PowerPC hardware, it no longer suffices in the modern world where an application might run in 32-bit or 64-bit mode and on both PowerPC and Intel hardware. The reason has to do with differences in the way structures are laid out in memory. It might not matter most of the time for values saved in the user defaults, because they usually stay on the machine where they were set. When you run Vermont Recipes on another computer, it will save its user defaults values in a format appropriate for that computer. However, a user might buy a new computer and transfer not only the application but the old user defaults files to the new computer, and you also have to anticipate the possibility of sharing user defaults values between computers across a network. A better way to save structures to user defaults, therefore, is to use a platform-agnostic technique like archiving or string conversion. String conversion is easiest: Just use paired functions like `NSStringFromSize()` and `NSSizeFromString()` to convert between them. To do this, replace the code above with this:

```
NSSize standardSize = [[self window] frame].size;
NSString *standardSizeString = NSStringFromSize(standardSize);
[[NSUserDefaults standardUserDefaults] setObject:standardSizeString
    forKey:VRDefaultDiaryWindowStandardSizeKey];
```

A side benefit to using the string conversion technique is that the preferences setting is now human readable. It reports the standard size as “{600, 922}.”

3. Finally, implement the `-windowWillUseStandardFrame:defaultFrame:` delegate method. In the `DiaryWindowController.m` implementation file, after the `-windowDidUpdate:` delegate method, insert this if you are still using the `NSData` code to save the standard size to the user defaults:

```
- (NSRect)windowWillUseStandardFrame:(NSWindow *)window
    defaultFrame:(NSRect)newFrame {
    NSSize standardSize;
    [[[NSUserDefaults standardUserDefaults]
        objectForKey:VRDefaultDiaryWindowStandardSizeKey]
        getBytes:&standardSize length:sizeof(NSSize)];
    NSRect frame = [window frame];
    frame.origin.y -= standardSize.height - frame.size.height;
    return NSMakeRect(frame.origin.x, frame.origin.y,
        standardSize.width, standardSize.height);
}
```

These statements use `NSData`’s `-getBytes:length:` method to extract the standard state `NSSize` structure’s bytes from the `NSData` object saved in the user defaults. They adjust the y-coordinate of the incoming window frame’s origin, subtracting the difference in window height between the standard height and

the current height, to pin the top-left corner in place. They then use the adjusted origin and the retrieved standard size to populate and return a new `NSRect`.

Again, this technique is fragile in the modern world. To change to a safer, platform-agnostic technique, use string conversion. Replace the first two statements in the previous code with this:

```
NSString *standardSizeString = [[NSUserDefaults standardUserDefaults]
    stringForKey:VRDefaultDiaryWindowStandardSizeKey];
NSSize standardSize = NSSizeFromString(standardSizeString);
```

4. Follow the same steps for the recipes window. This works no matter how many recipes windows the application can open, as long as you set only the size and pass the incoming position through unchanged.

At the top of the `RecipesWindowController.h` header file, just before the `@interface` directive, enter this declaration:

```
extern NSString *VRDefaultRecipesWindowStandardSizeKey;
```

At the bottom of the `RecipesWindowController.m` implementation file, after the `@end` directive, define it like this:

```
NSString *VRDefaultRecipesWindowStandardSizeKey =
    @"recipes window standard size";
```

5. Next, in the `-windowDidLoad` delegate method that you wrote in the `RecipesWindowController.m` implementation file in Recipe 5, add code to save the newly opened window's size to the user defaults, at the end. Use the string conversion technique, like this:

```
NSSize standardSize = [[self window] frame].size;
NSString *standardSizeString = NSStringFromSize(standardSize);
[[NSUserDefaults standardUserDefaults] setObject:standardSizeString
    forKey:VRDefaultRecipesWindowStandardSizeKey];
```

6. Finally, implement the `-windowWillUseStandardFrame:defaultFrame:` delegate method using the string conversion technique. In the `DiaryWindowController.m` implementation file, after the `-windowDidUpdate:` delegate method, insert this:

```
#pragma mark DELEGATE METHODS
```

```
NSString *standardSizeString = [[NSUserDefaults standardUserDefaults]
    stringForKey:VRDefaultRecipesWindowStandardSizeKey];
NSSize standardSize = NSSizeFromString(standardSizeString);
NSRect frame = [window frame];
frame.origin.y -= standardSize.height - frame.size.height;
return NSMakeRect(frame.origin.x, frame.origin.y,
    standardSize.width, standardSize.height);
```

The `-windowWillUseStandardFrame:defaultFrame:` implementations for the two windows are nearly identical and could be consolidated in some common method. However, you may eventually want to customize the zoom behavior of one of them but not the other, so I think it's best to implement them separately.

7. Build and run the application. The recipes window opens automatically, centered horizontally and positioned up against the menu bar. Click its zoom button, and then click it again. Nothing happens. This is the correct behavior according to the *HIG*, because you have not yet resized the window to establish a user state. Resize it now, making it larger or smaller, and move it somewhere else on the screen. Now click the zoom button again. The window resizes to its initial size. If it was in the central area of the screen after you resized it, zooming it back to the initial size left its top-left corner anchored in place. Try resizing it to a small user state, and move it beyond the bottom-left or -right corner of the screen so that it is mostly off screen at the bottom and to one side or the other and so that it overlaps the Dock. Click the zoom button, and see that it moves fully onscreen when it resizes to the standard state. Zoom again, and watch it return to its mostly offscreen size and position. Try the same experiments with the diary window.

Step 4: Autosave the Position and Size of the Document Windows

According to the *Apple Human Interface Guidelines*, when a previously saved document is reopened, its window should open in the same place and with the same size that it had when it was last closed, perhaps with some adjustments if the user's display arrangement has changed in the meantime. This does not happen automatically.

You might think that the easiest way to fix this is to fill in the document's Autosave field in the Window Attributes inspector in Interface Builder, but you'll find that this yields unsatisfactory results. Trash the saved diary document, and then try an experiment. Type a name like **diary window** in the Autosave field of the inspector, save the nib file, and build and run the application. Create a new Chef's Diary document, drag the window away from its initial position, make it as small as possible, and then close it without saving it. Now open another new Chef's Diary document. Its window opens where you left the first one, and it has the same small size. This isn't right because the new document isn't the same Chef's Diary document that you just closed. It's a new, empty document, and the *HIG* specifically requires that new documents open centered horizontally near the top of the window.

The Interface Builder Autosave field is perfectly good for simple applications having a single standard main application window. It is not appropriate for Vermont Recipes. Here, you chose to use the document-based application template to take advantage of the many benefits of `NSDocument`, while nevertheless limiting it to a one-of-a-kind current diary document. In a typical document-based application, documents' differing window states are saved in the documents themselves. Here, with only one current diary document, you rely on autosaving instead of saving window state in the document. You'll have to write some code to make this work.

Apple's documentation for autosaving window frames programmatically is pretty basic, and it doesn't go into much detail regarding usage of the available methods. Perhaps for this reason, the developer mailing lists and discussion forums have reflected confusion over a period of several years. For much of this time, a notorious bug—finally fixed in Leopard—made it effectively impossible to use Interface Builder's Autosave field in applications that rely on `NSWindowController`. Developers were lost, having no way to use Interface Builder's Autosave field and no instructions for autosaving a window's frame programmatically. Another issue has been the absence of documentation regarding the difference, if any, between the two similarly named methods that `NSWindow` and `NSWindowController` implement, `-setFrameAutosaveName:` and `-setWindowFrameAutosaveName:`, respectively.

You can, if you wish, avoid autosaving a window's frame and save the position and size manually, so to speak, without using either of the methods for setting an autosave name. Simply call `NSWindow`'s `-saveFrameUsingName:` method in appropriate places to save the window's current frame, and call `-setFrameUsingName:` to set the position and size of a reopened window accordingly.

However, once you know how to use the automatic behavior of the window frame autosave system, you will find that it is easier than saving the frame manually. The secret for making it work in a complex application is the realization that a window forgets a window's frame autosave name when the user closes the window or quits the application. You therefore have to set it more than once, when the user saves the document and again later, when the document reopens itself or the user reopens the document.

1. Start by defining a global name for the autosave name string. For the diary window, call it `VRDiaryWindowAutosaveName`.

In the `DiaryWindowController.h` header file, declare it near the top, just after the declaration of `VRDefaultDiaryWindowStandardSizeKey`, like this:

```
extern NSString *VRDiaryWindowAutosaveName;
```

At the end of the `DiaryWindowController.m` implementation file, define it like this:

```
NSString *VRDiaryWindowAutosaveName = @"diary window";
```

When the window's frame is saved in user defaults, the full key for the frame string's value will be `NSUserDefaultsFrame diary window`.

2. Now set the diary window's autosave name and save the window's frame to the user defaults whenever the user explicitly saves the document.

The best way to respond to the user's saving the diary document is by overriding the `-saveToURL:ofType:forSaveOperation:error:` method in `DiaryDocument`, which you did in Recipe 6. It is important to override this method, because it allows you to distinguish between user-initiated save operations and automatic save operations. You're only interested in user-initiated saves for present purposes, because automatic saves are invisible to the user. Automatic saves occur even before the user has saved a document, saving it invisibly to a special location under a special name if the document doesn't yet have a user-supplied name, so that the application can restore unsaved work in the event of a power failure or other accident. From the user's perspective, an autosaved document may not have been saved at all, and it remains possible to create a new, empty diary document. You should therefore set the window frame's autosave name and save the window's frame only when the user saves the document explicitly. The `-saveToURL:ofType:forSaveOperation:error:` method lets you do that.

The document is supposed to focus on its data, the MVC model, not on how the data is displayed in the user interface, the MVC view. You therefore should not place any code in the `-saveToURL:ofType:forSaveOperation:error:` method that actually manipulates the window.

A traditional way to separate the model from the view in these circumstances is to use the Cocoa *notification center*. Notifications are similar to delegate methods, in that they offload functionality from one class to another in a way that preserves flexibility, but they differ in their generality. A class can have only one delegate and thus only one class that can partner with it, so the delegate design pattern imposes a somewhat formalized structure on the overall layout of your code. Notifications, by contrast, are broadcast, in a manner of speaking, and any class can watch for them and respond. They are also just a little bit easier to implement. You simply post a notification to the application's notification center, leaving it up to other classes to decide whether to observe them and respond. Notifications are sent and observers respond to them synchronously, so the user interface is updated immediately. Read the "Notifications" sidebar for more information.

Notifications

Cocoa creates for you a notification center, which an application can use to post and send notifications to any other objects that register to observe them.

The notification technique differs from delegation in several respects and is useful in different situations. For one thing, multiple objects can register to receive a single notification; that is, an object can have many observers, whereas it can have only one delegate. Notifications are therefore useful when an object does something that affects the application in a way that many other objects need to know about and respond to—for example, to synchronize their state with that of the sending object.

For another, notification does not require the notifier to know anything about the observer or even that there are any observers, and an observer need not know anything about the notifier. They only need to share knowledge of the existence and nature of the notification mechanism. This makes notification a more flexible technique than delegation. For example, a developer can add new functionality to an application without altering the notifier in any way or even knowing anything about it. It is not necessary for an observer to have access to a notifier through an instance variable or accessor method, as a delegate must in order to become the notifier's delegate; it is necessary only to register with the notification center to observe the notification. Optionally, notifications can include information about the notifying object that observers can use to understand the notification in greater detail.

A limitation of notifications is that, unlike delegates, an observer cannot interfere in any way with the notifying object. The observer cannot, for example, prevent the event that is the subject of the notification from happening, as some delegate methods can. Also, the observer cannot return information to the object that posted the notification.

When you post a notification, you do it using a notification name that you define globally so that observers can discriminate between notifications to observe. You usually include the posting object, self, or some other object, so that observers have a reference to it that allows them to access features of the posting object, but including the object is optional. You can also include, optionally, a userInfo dictionary containing any information the observer might need—for example, information about the state the posting object is in at the time of posting.

There is some processing overhead associated with notifications, but in general they are very efficient and can be used in most situations without concern regarding performance. Cocoa itself makes extensive use of notifications.

(continues on next page)

Notifications *(continued)*

Notifications that use `NSNotificationCenter` are task specific, or local. A notification you post in an application can be observed only by other objects within that same application. You access the application's notification center using the class method `+[NSNotificationCenter defaultCenter]`. For cross-process notifications, use `NSDistributedNotificationCenter`.

Notifications that use `NSNotificationCenter` are also synchronous. For asynchronous notifications, use `NSNotificationQueue`.

Enter this statement, in the `saveToURL:ofType:forSaveOperation:error:` method:

```
[[NSNotificationCenter defaultCenter] postNotificationName:
    VRDidSaveDiaryDocumentNotification object:self];
```

This uses the `VRDidSaveDiaryDocumentNotification` variable, which you must declare and define globally. You could have named it more generally as `VRUserDidSaveDocumentNotification` and use it with the recipe document as well, requiring observers to detect which kind of document was saved by examining the sending object. Here, however, you have not yet made any decisions about how the recipes document will save its data, so focus on the diary document and use the more specific notification name.

In the `DiaryDocument.h` header file, declare the notification name variable near the top, above the `@interface` directive, like this:

```
extern NSString *VRDidSaveDiaryDocumentNotification;
```

At the end of the `DiaryDocument.m` implementation file, define it like this:

```
NSString *VRDidSaveDiaryDocumentNotification =
    @"did save diary document notification";
```

You already imported `DiaryDocument.h` into the `DiaryWindowController.m` implementation file in Recipe 3, so you will be able to use this variable in the window controller too.

Now arrange for the `DiaryWindowController` object to observe and respond to the notification. There are two basic requirements: The window controller must register to observe the notification, and it must declare and implement a method to be called every time the notification is received. The first requirement, registering as an observer, carries with it an ancillary obligation to remove the registration before the window controller is deallocated.

To register, add these statements to the `-windowDidLoad` method at the end of the `DiaryWindowController.m` implementation file:

```
NSNotificationCenter *defaultCenter =
    [NSNotificationCenter defaultCenter];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(didSaveDiaryDocument:)
    name:VRDidSaveDiaryDocumentNotification object:nil];
```

You passed `nil` in the `object` parameter in the second statement. As a result, the window controller will receive every notification having the specified name. You could have specified the sender of the notification in the `object` argument, which would have limited the notifications received to those with the specified name that also include the specified object as sender. It would have been important to do that if you had named the notification generally as `VRUserDidSaveDocument Notification`, to ensure that the `DiaryWindowController` would have received notifications having this name only from the `DiaryDocument`, not those from the `RecipesDocument`. You could even have registered using `nil` as the notification name, to receive all notifications posted by the diary document. Here, specifying the sending object is not important because the name of the notification is sufficient to identify the sender.

You chose to name the selector `didSaveDiaryDocument:`. Now you must implement a method with that signature. It is not necessary to declare the method, because Cocoa calls it by its selector, which you just registered with the notification center. Many developers nevertheless declare a method like this in case they eventually do need to call it, and in this case you will discover in Recipe 8 that you do need to declare it after all. Add the method to the `DiaryWindowController.m` implementation file, after the `-windowWillUseStandardFrame:defaultFrame: delegate` method, as follows:

```
#pragma mark NOTIFICATION METHODS

- (void)didSaveDiaryDocument:(NSNotification *)notification {
    [[self window] setFrameAutosaveName:VRDiaryWindowAutosaveName];
    [[self window] saveFrameUsingName:VRDiaryWindowAutosaveName];
}
```

This looks just like a typical delegate method, and it behaves in much the same way, executing whenever the specified event occurs. It sets the window's frame autosave name and saves the window's frame to the user defaults every time the user saves the Chef's Diary. It does so where this code belongs, in the window controller, not in the document. If you felt it more appropriate to place it at the document controller or application level, you could easily do so, because the document broadcast the notification at large and any object can observe it.

Remember to remove the observer before deallocating the window controller. In the `DiaryWindowController.m` implementation file, at the end of the Delegate Methods section, add this method:

```
- (void>windowWillClose:(NSNotification *)notification {
    NotificationCenter *defaultCenter =
        [NSNotificationCenter defaultCenter];
    [defaultCenter removeObserver:self];
}
```

3. Now return to the task at hand, autosaving and restoring the diary window's frame. Set the window's autosave name again whenever the window is reopened. This should happen whether the user is opening the document from the Vermont Recipes File menu using the Open Chef's Diary, Open, or Open Recent menu items; or from the Finder by double-clicking the document's icon, by selecting it and choosing File > Open, or by dropping it on the Vermont Recipes application icon; or from AppleScript or some other application that knows how to open files.

You have to set the autosave name again because the application forgets it when the user closes the document or quits the application. If the user then reopens the saved document or relaunches the application by opening a saved document, the application is no longer aware that the user defaults contained saved information regarding the diary window's frame. Setting the autosave name again when the user reopens the document solves that problem.

A good place to set the autosave name is in the `-windowDidLoad` method in `DiaryWindowController`. The application then automatically configures the window's frame after `-windowDidLoad` executes.

At the end of the existing `-windowDidLoad` method in the `DiaryWindowController.m` implementation file, add these statements:

```
VRDocumentController *controller =
    [VRDocumentController sharedDocumentController];
if ([controller canOpenURL:[controller currentDiaryURL]]) {
    [[self window] setFrameAutosaveName:VRDiaryWindowAutosaveName];
}
```

You test whether the document has been saved and is not in the Trash by using the document controller's `-canOpenURL:` method, which you wrote in Recipe 6. This ensures that the autosave name is not set prematurely, when the user creates a new diary document that has not previously been saved.

4. Explore how well this works. You need to take some preliminary steps in order to conduct a valid test. First, open the `~/Library/Preferences` folder in the Finder and locate the `com.quecheesoftware.Vermont-Recipes.plist` file, and then drag

it to the Trash. This is required in order to start with a blank slate in terms of the information that is stored in the user defaults. Second, clean the project by choosing Build > Clean, and build it using the Release configuration, not the Debug configuration. The Finder may not care, but I feel more confident testing a release build when I'm working outside Xcode.

Now run the application. The recipes window opens centered and near the top of the screen. Choose File > New Chef's Diary. A new, empty diary window appears, and it, too, is centered near the top of the screen.

Drag the diary window lower and toward the left edge of the screen, and make it smaller; then close it. Choose File > New Chef's Diary again. A new diary window appears, and it is once again centered near the top of the screen. It correctly ignored the position and size that the previous diary window had when you closed it without saving it.

This second empty diary document is named Untitled 2. A Cocoa document-based application handles the naming rules prescribed by the *HIG* correctly without additional effort on your part. Drag this window down and toward the left, and make it smaller. This time, save it before closing it. Choose File > Save As, give it a name like **Test**, and save it to the Desktop; then close it. Choose File > Open Chef's Diary, and the Test document's window reopens, this time right where it was when you closed it, and with the same size.

Close it again, and this time open it by double-clicking it in the Finder, or by selecting it and choosing File > Open in the Finder, or by dragging it onto the Vermont Recipes icon in the Dock. Again, it reopens at the same position and with the same size.

Drag it over to the right side of the screen and make it as large as you can; then close it. Reopen it using any of the available techniques, and it opens in the same place and just as large.

Now make it as small as you can and drag it to the lower center of the screen, just above the Dock. This time, quit Vermont Recipes by choosing Quit from its application menu. Now double-click the Test document's icon in the Finder. Vermont Recipes launches, and the document window opens where you left it, toward the bottom center of the screen and sized as small as it can be.

Perform another test. Drag the Test document's icon into the Trash, but don't empty the Trash. Now, in Vermont Recipes, choose File > New Chef's Diary and, if you like, move the new, empty document window somewhere else on the screen and change its size; then close it without saving it. Now drag the Test document's icon out of the Trash back onto the Desktop, and double-click it or choose File > Open Chef's Diary. The Test document has been resurrected, and

its window opens centered toward the bottom of the screen, as small as it can be, just as you left it before dragging it to the Trash.

You should also perform some tests that involve dragging the window most of the way offscreen—for example, below the bottom behind the Dock (assuming your Dock is positioned at the bottom of the screen), or so that it straddles two displays. Then close it and reopen it. You see that Cocoa automatically repositions it so that it is fully on one of the screens and not obscured behind the Dock, much as Cocoa handled zooming the document window.

There are more tests you could perform, such as renaming the Test document, closing it, and moving its icon into another folder. They all work, and you can be satisfied that you have successfully completed this step.

It would be premature to port this code to the recipes window at this time, because it depends in part on code that saves the document, at least to post a notification. You haven't yet addressed the contents of the recipes document, so you shouldn't write any code to save it at this time.

Step 5: Autosave the Position of the Divider in the Diary Window

Leopard brought a long-requested new feature to NSSplitView, the ability to autosave the position of the divider. The *Apple Human Interface Guidelines* haven't caught up. They provide no guidance on what to do with the split view in a new, empty window. Fill this gap by applying the same rule that you applied to the frame of a new, empty document's window frame: A divider's position should not be autosaved and restored when creating a new diary document, but only when opening a saved diary document.

Before writing any code, experiment with the application while relying solely on Interface Builder to set the autosave name. Open the DiaryWindow nib file and select the split view divider. In the Split View inspector, enter **diary split view** in the Autosave field. Save the nib file, trash any saved diary document icons left over from Step 4, and then build and run the application. Choose File > New Chef's Diary and, in the empty diary window, drag the divider partway up. Close the document and create another new Chef's diary document. The divider is right where you left it. This is inappropriate, since a new document window should have a standard appearance every time the user creates one. Plainly, as with the window frame autosave feature, you should use Interface Builder to implement the split view divider autosave feature only in simple applications where a main application window contains a split view.

For more refined behavior, take the same approach to resolving this issue that you took with autosaved window frame names in Step 4. First, remove the Autosave name you just experimented with in the `DiaryWindow` nib file and save the nib file. Then set up a global string for the divider autosave name in code, and arrange to implement this feature.

1. In the `DiaryWindowController.h` header file, declare the autosave name string near the top, just after the declaration of `VRDiaryWindowAutosaveName`, like this:

```
extern NSString *VRDiarySplitViewAutosaveName;
```

At the end of the `DiaryWindowController.m` implementation file, define it like this:

```
NSString *VRDiarySplitViewAutosaveName = @"diary split view";
```

2. In order to carry out this task, you need an outlet for the split view.

In the `DiaryWindowController.h` header file, declare the outlet between the brackets following the `@interface` directive, after the `searchField` outlet, like this:

```
IBOutlet NSSplitView *splitView;
```

Declare its accessor method after the `-searchField` accessor method, like this:

```
- (NSSplitView *)splitView;
```

Define the accessor method in the `DiaryWindowController.m` implementation file, like this:

```
- (NSSplitView *)splitView {  
    return [[splitView retain] autorelease];  
}
```

Finally, connect the new outlet in Interface Builder by Control-dragging from the File's Owner proxy to the split view divider in the diary window design surface and selecting the `splitView` outlet.

3. `NSSplitView` doesn't declare a method named like `NSWindow`'s `-saveFrameUsingName:` method. However, `NSSplitView`'s `-adjustSubviews` method performs a similar role.

As with the window's frame, you don't want to set the autosave name when the user creates a new, empty document. Instead, wait until the user saves the document and set it at that time. Add these two statements to the end of the `-didSaveDiaryDocument:` notification method you just wrote in the `DiaryWindowController.m` implementation file:

```
[[self splitView] setAutosaveName:VRDiarySplitViewAutosaveName];  
[[self splitView] adjustSubviews];
```

The first statement sets the autosave name, but it does not actually save the current state of the split view divider to the user defaults. If you did nothing more, the state of the divider would not be available for restoration when the user reopened the window. The divider position would be saved only if the user happened to reposition the divider after saving the document. You could presumably force the divider's position to be saved by programmatically repositioning the divider, but the visual effect would be distracting. By experimentation, it is clear that calling `NSSplitView's -adjustSubviews` method accomplishes the desired saving of the divider's position without visual distraction.

This saves the split view's divider position in the user defaults as an array containing two strings, each of which encodes the frame of one of the split view panes and the word *YES* or *NO*, depending on whether the pane is collapsed. The meaning of *YES* or *NO* is documented only in the `NSSplitView.h` header file. The full key will be `NSSplitView Subview Frames diary split view`.

4. You have to set the autosave name again in the window controller's `-windowDidLoad` method, because the application forgets it when the user closes the document or quits. As with the window's frame, you must of course avoid setting the autosave name if the diary document is currently unsaved or, if saved, is in the Trash. Otherwise, the divider's position would be saved and restored even when the user created new diary documents. You already handled this for the window's frame by testing for `-canOpenURL:` in `-windowDidLoad`. To complete your code for autosaving the split view divider, simply add this statement to the end of the `if` block in `-windowDidLoad`:

```
[[self splitView] setAutosaveName:VRDiarySplitViewAutosaveName];
```

5. Perform the same tests you performed in Step 4 to confirm that the split view divider behaves as specified.

Step 6: Autosave the Recipes Document's Toolbar Configuration

In Step 2 of Recipe 2, you added a toolbar to the recipes window and set it up so that the user can customize the toolbar by adding and removing toolbar items. You will not complete development of the recipes window in this book, but as long as you're thinking about autosaving, it would be a good idea to autosave any custom configuration of the toolbar now. If you don't do this, the customized toolbar will revert to its default configuration the next time the user launches the application, and you will be faced with a confused and unhappy user.

This is really simple, as autosaving goes. Open the `RecipesWindow.nib` file in Interface Builder, select the toolbar, and, in the Toolbar Attributes inspector, select the Autosaves Configuration checkbox. Save the nib file, rebuild and run the application, and test it. Run it once, choose View > Customize Toolbar, and remove and add a few toolbar items. When you're done, quit and relaunch the application. The custom toolbar configuration reappears. If you had performed this test before setting the autosave feature, the toolbar would have reverted to its default configuration.

Step 7: Autosave the Diary Document's Contents

Your users will thank you for making the diary document save its contents periodically. Most of them don't save their work nearly as often as they should, and they are sure to lose important data sooner or later in a power outage or some other accident.

You turn on periodic autosaving for all of the application's documents by calling an `NSDocumentController` method, `-setAutosavingDelay:`, to set the autosaving delay to a time interval greater than zero. The autosaving delay is `0.0` by default, which means that autosaving is turned off. Both `NSDocumentController` and `NSDocument` declare a number of methods you can use to customize autosaving if you wish—for example, by preventing a specific kind of document or a specific document from autosaving at all.

When autosaving is turned on, the application notices when the user makes a change to a document's contents, and it then waits the specified time interval. If by the time the interval has expired the user has not saved the document explicitly, Cocoa autosaves it. If the user has never saved the document explicitly, Cocoa saves it in a special location under a special name. By default, the location is `~/Library/Autosave Information`, but you can change this if you have a reason to do so. The name, the first time the diary document is autosaved, is *Unsaved Vermont Recipes Document*. It is saved along with a property list file that contains a bookmark and a timestamp. Autosaving also occurs periodically after a document has been saved, of course. The property list file stays in the Autosave Information folder, but the autosaved document is saved by default in the same folder where the document itself was saved, under the name of the saved document plus (*Autosaved*). When the user next saves the document explicitly, or deletes it, the autosaved document and the property list file are deleted automatically.

From the user's point of view, a document that has been autosaved but never saved explicitly has not been saved at all. There is no document icon on the desktop or anywhere else the user is likely to look; and in the case of the diary document, for

example, the user can still create a new, empty document by choosing File > New Chef's Diary. The user cannot open an existing document, because there is no saved current Chef's Diary. The user cannot perform any standard actions to open the autosaved document, because it is meant to be invisible to the user.

The useful action occurs if the application crashes or is terminated irregularly—for example, because of a power outage. When the application is next launched, Cocoa automatically restores the document's contents to those that were last autosaved, and it opens them in a new window. It offers no explanation; the window simply opens when the user launches the application and displays the last autosaved contents.

When an autosaved document's contents are restored, that's all that is restored unless you do some more work. The window frame and, in the case of the diary document, the position of the split view divider, are not restored at the same time. In Steps 4 and 5, you arranged to save these user interface states to the user defaults when the user saves the document. Once you have basic document autosaving working, you will go on to implement some of these other autosave methods to restore the position and size of the window and the divider when the document's contents are restored as well.

1. Start by turning on autosaving. A good place to do this is in `VRApplicationController`, which you created in Recipe 5 to implement a couple of action methods for application-wide menu items. At that time, you noted that `VRApplicationController` can also serve as the application's delegate, and you connected its `delegate` outlet in Interface Builder. Now you're ready to make use of its capabilities as the application's delegate.

`NSApplication` declares many delegate methods that give you the opportunity to customize how an application responds to a variety of events. One of the most common events to monitor is launching the application. It is convenient to carry out many setup operations as soon as the application has completed the launch process and is ready for use. To do this, you implement `NSApplication's -applicationDidFinishLaunching:` delegate method in the application's delegate. That's a good place to turn on document autosaving, since it is an application-wide feature.

In the `VRApplicationController.m` implementation file, add this method at the end:

```
#pragma mark DELEGATE METHODS

- (void)applicationDidFinishLaunching:(NSNotification *)notification {
    [[VRDocumentController sharedDocumentController]
        setAutosavingDelay:5.0];
}
```

This causes the application to autosave the diary document or the recipes document 5 seconds after the user makes any change to its contents, unless the user explicitly saves changes before that interval expires or takes some other action that makes autosaving inappropriate, such as closing the document. Five seconds may seem a little overeager, but it is convenient for testing during development. You will eventually turn the autosave delay interval into a user-settable preference, much as TextEdit and many other applications do.

2. Perform a little experiment to see what happens when the application crashes or the power fails. Instead of going outside and dropping a tree across your power lines, simply force-quit the application after making some changes to the document and letting them be autosaved.

To perform the experiment, launch the application in Xcode—launching it in Debug mode is fine. Create a new diary document, and type some text into it. A little over 5 seconds later, click the red Tasks button in the toolbar of an Xcode editing window or the Build Results window. This kills the application in a manner that is equivalent to force-quitting or pulling the power plug.

Now relaunch the application. The diary window reopens automatically, and the text you just typed is intact. The window's title indicates that it is an unsaved document, but your changes were preserved and restored.

Close the window, and then click Don't Save when an alert asks whether you want to save changes. Choose File > New Chef's Diary, and a new, empty diary window opens. This is as it should be, since you did not save the restored document when you closed its window.

3. You could stop here. Except for the user-settable preference you will implement later, document autosaving plainly works. However, try a similar experiment, and you see that you need to do a little more work.

As before, type some text into the new, empty diary document. This time, also make the window smaller and move it elsewhere on the screen, and drag the split view divider up toward the middle of the window. Then, after a little more than 5 seconds, click the red Tasks button again to kill the application. Relaunch it, and the window again reopens with the new text restored. However, the window is in the standard state, centered near the top of the screen and at its initial size, not where you left it when you pulled the plug. Similarly, the split view divider is not where you left it when you pulled the plug.

For a consistent user experience, restoring an autosaved document should not only restore the text that the user last worked on, but also restore the size and position of the window and the position of the divider when the user last worked on it. This is not a new, empty document, but an existing document, even though the user has never saved it.

4. To achieve the desired goal, you have to write some more code. First, you must set the window frame autosave name and the split view name you worked with in Steps 4 and 5, and save the values, on one more occasion. Previously, you set the autosave names and saved the values to the user defaults when the user saved the document. Now, you must also do this every time the system autosaves the document. Second, in a moment, you must also arrange to set the autosave names again when the document is reopened, even if it has not previously been saved, if it is opening in an autosave restoration operation and is about to display the autosaved changes. As you already know, the application forgets the autosave names after a crash, so you must reset them when the user relaunches the application.

Since you have now decided to set the autosave names both for user-initiated save and autosave operations, you could consolidate the code you wrote in Steps 4 and 5 to handle both at once. However, by maintaining separate code paths, you preserve your freedom to customize autosave behavior later.

Start by modifying the `-saveToURL:ofType:forSaveOperation:error:` method in the `DiaryDocument.m` implementation file, which you originally wrote in Recipe 6 and modified in Step 4 of this recipe. Break the `if` block into pieces to first test whether the save operation was successful, and then, depending on whether this is a user-initiated save or an autosave operation, post an appropriate notification.

```
if (success) {
    if (saveOperation == NSAutosaveOperation) {
        [[NSNotificationCenter defaultCenter] postNotificationName:
            VRDidAutosaveDiaryDocumentNotification object:self];
    } else {
        [[VRDocumentController sharedDocumentController]
            setCurrentDiaryURL:absoluteURL];
        [[NSNotificationCenter defaultCenter] postNotificationName:
            VRDidSaveDiaryDocumentNotification object:self];
    }
}
```

5. You must create a new `VRDidAutosaveDiaryDocumentNotification` string variable for this purpose. At the top of the `DiaryDocument.h` header file, after the existing notification string variable, add this:

```
extern NSString *VRDidAutosaveDiaryDocumentNotification;
```

At the end of the `DiaryDocument.m` implementation file, add this:

```
NSString *VRDidAutosaveDiaryDocumentNotification =
    @"did autosave diary document notification";
```

6. In the `DiaryWindowController.m` implementation file, add a statement to register the window controller as an observer of the new notification at the end of the existing `-windowDidLoad` method, like this:

```
[defaultCenter addObserver:self
    selector:@selector(didAutosaveDiaryDocument:)
    name:VRDidAutosaveDiaryDocumentNotification object:nil];
```

You have already taken care of removing self as an observer in the `-windowWillClose:` method.

7. Finally, provide for the actual work of the notification by implementing the `-didAutosaveDiaryDocument:` method that is triggered by the notification. Add this method to the `DiaryWindowController.m` implementation file immediately following the `-didSaveDiaryDocument:` method:

```
- (void)didAutosaveDiaryDocument:(NSNotification *)notification {
    [[self window] setFrameAutosaveName:VRDiaryWindowAutosaveName];
    [[self window] saveFrameUsingName:VRDiaryWindowAutosaveName];
    [[self splitView] setAutosaveName:VRDiarySplitViewAutosaveName];
    [[self splitView] adjustSubviews];
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

The work of this method is familiar from the `-didSaveDiaryDocument:` method you wrote earlier. It sets the autosave names and saves the values to the user defaults as protection against that rainy day when the power fails.

The difference from the `-didSaveDiaryDocument:` method is that `-didAutosaveDiaryDocument:` also forces the application to write the user defaults to disk immediately in case there is an unexpected crash right away, using the `NSUserDefaults -synchronize` method. For efficiency, an application's user defaults settings are normally kept in memory, and then written to disk automatically from time to time and when the application quits. The *NSUserDefaults Class Reference* explains that you can nevertheless call `-synchronize` yourself if you need to ensure that values are written to disk immediately. Safeguarding values so that they survive an unexpected crash seems like an appropriate time to call `-synchronize` explicitly. After all, you're going to the trouble of capturing and saving the values, and it isn't much additional trouble to go all the way and write them to disk immediately.

Another reason to write these values to disk immediately is to facilitate testing. The developer mailing lists are full of complaints that restoring autosaved values only works episodically. The reason for inconsistent results during testing is likely that autosaved values were sometimes written to disk and sometimes not, depending on the vagaries of the system's internal schedule for calling `-synchronize`. Calling it explicitly generates completely consistent test results.

8. All that's left is to set these autosave names again when the user relaunches the application and the system reopens the document after restoring the autosaved document contents. The trick is to do this only if the document is being restored after a crash. You can't make this distinction in `DiaryWindowController`'s `-windowDidLoad` method, where you set the autosave names again when the user reopens a saved window, without doing some more coding, because there is no `-isRestoringAutosavedDocument` method in `NSDocument` to distinguish between loading a window while the user opens a document and loading a window while the application restores an autosaved document. You'll have to write your own.

One way to do this would be to override `NSDocumentController`'s `-reopenDocumentForURL:withContentsOfURL:error:` method in `VRDocumentController`. In the "Autosaving in the Document Architecture" section of *Document-Based Applications Overview*, Apple identifies this as a method to override in order to customize the restoration operation.

In this case, however, it is easier to use another hook into the restoration process, `NSDocument`'s `-initWithURL:withContentsOfURL:ofType:error:` method. This is an ordinary object initialization method, called by the system to initialize the new document that is opened when restoring the document's autosaved contents. It is easier in the sense that the system has already determined for you that this is a `DiaryDocument`; otherwise, if you used the `-reopenDocumentForURL:withContentsOfURL:error:` method, you would have to test for that condition yourself.

Override the `-initWithURL:withContentsOfURL:ofType:error:` method at the top of the `DiaryDocument.m` implementation file, like this:

```
#pragma mark INITIALIZATION

- (id)initWithURL:(NSURL *)absoluteDocumentURL
    withContentsOfURL:(NSURL *)absoluteDocumentContentsURL
    ofType:(NSString *)typeName error:(NSError **)outError {
    if ((self = [super initWithURL:absoluteDocumentURL
        withContentsOfURL:absoluteDocumentContentsURL
        ofType:typeName error:outError])) {
        isRestoringAutosavedDocument = YES;
    }
    return self;
}
```

The only thing you did in the override method was to set an instance variable, `isRestoringAutosavedDocument`, to YES. You must declare the instance variable

and write accessor methods for it. Generally, in an initialization method, you should not call the set accessor method but instead set the value of the instance variable directly, because some setter methods have side effects that might be inappropriate at initialization time. You will use the setter later to reset the instance variable to NO.

In the `DiaryDocument.h` header file, declare the new instance variable after the `diaryDocTextStorage` instance variable, like this:

```
BOOL isRestoringAutosavedDocument;
```

At the end of the Accessor Methods section of the header file, declare the two accessor methods:

- (void)setIsRestoringAutosavedDocument:(BOOL)flag;
- (BOOL)isRestoringAutosavedDocument;

Here, you included *is* in both the getter and the setter. Apple’s *Coding Guidelines for Cocoa* are ambiguous about the use of *is*, and they do not spell out the full range of possibilities that are available. The *Guidelines* recommend that, if you don’t use *is* in the instance variable name, you should leave it out of the setter’s name as well, but nevertheless include it in the getter’s name. If you think about it, you see that *is* in the getter often helps to clarify that you are getting the value of an adjective rather than a noun. Here, you avoid ambiguity by naming the instance variable itself with *is*, and the normal rule that a getter and setter should reflect the name of the instance variable therefore applies.

In case you’re interested, another discussion of *is* appears in the “Accessor Search Implementation Details” section of the *Key-Value Coding Programming Guide*. There, under “Default Search Pattern for valueForKey:,” Apple spells out how Cocoa searches for accessor methods or instance variables when a program uses key-value coding (KVC). KVC first searches for a getter method named either `-get<Key>`, `-<key>`, or `-is<Key>`, in that order. If it finds no such getter, it searches for an instance variable named `_<key>`, `_is<Key>`, `<key>`, or `is<Key>`. Similar rules apply to setters. In short, it is legal to include *is* in the instance variable, the getter and the setter, or any or none of them, in virtually any combination.

Define them in the `DiaryDocument.m` implementation file:

- ```
- (void)setIsRestoringAutosavedDocument:(BOOL)flag {
 isRestoringAutosavedDocument = flag;
}

- (BOOL)isRestoringAutosavedDocument {
 return isRestoringAutosavedDocument;
}
```

Now it's easy to modify `DiaryWindowController`'s `-windowDidLoad` method to handle restoration of autosaved document contents in addition to normal document opening operations. Simply change the `if` test at the end of the method to include a test for your new `isRestoringAutosavedDocument` instance variable, and reset the instance variable to `NO` because it has done its job. Here is the revised `if` block:

```
if ([controller canOpenURL:[controller currentDiaryURL]]
 || [[self document] isRestoringAutosavedDocument]) {
 [[self window] setFrameAutosaveName:VRDiaryWindowAutosaveName];
 [[self splitView] setAutosaveName:VRDiarySplitViewAutosaveName];
 [[self document] setIsRestoringAutosavedDocument:NO];
}
```

Immediately afterward, Cocoa reconfigures the position and size of the window and the position of the divider according to the autosaved values, and displays the window.

You did not call `-synchronize` here, just as you did not call it in the `-didSaveDiaryDocument:` method. You only called it in `-didAutosaveDiaryDocument:`, where you were preparing for a crash. You could call `-synchronize` anywhere, and the only cost would be increased disk accesses. I prefer to let Cocoa call it periodically behind my back apart from exceptional circumstances.

9. Restoring the autosaved diary document now works. However, it works without explaining to the user what has happened. When the restored document opens in its window on the screen after the user relaunches the application following a crash or a power failure, it just appears. There is no message to the user, and the title is no different. This is exactly how `TextEdit` behaves, and, for example, `Microsoft Word` behaves the same way except that the window's title includes "(Recovered)." The *HIG* provides no guidance in this area.

My personal experience is that the restored document can come as a surprise to the user, especially if some time has gone by since the power outage—power outages tend to last a long time where I live. The application does not normally open the Chef's Diary unless the user opens it on purpose, so why has it suddenly opened by itself this time? I think you should go ahead and present an alert when this happens, but give the user an opportunity to turn off future alerts if they become a nuisance.

To do this, use a different hook into the autosaved document restoration process. You just used the `-initWithURL:withContentsOfURL:ofType:error:` method, because it let you into the restoration process near the beginning, before the restored document is displayed. Now you want to get into the process after it has completed and the restored document is already visible on the screen. To do this, override `NSDocumentController`'s `-reopenDocumentForURL:withContentsOfURL:error:` method, which you considered earlier.

In the `VRDocumentController.m` implementation file, after the `-openDocumentWithContentsOfURL:display:error:` method, insert this method:

```
- (BOOL)reopenDocumentForURL:(NSURL *)absoluteDocumentURL
 withContentsOfURL:(NSURL *)absoluteDocumentContentsURL
 error:(NSError **)outError {
 BOOL success = [super reopenDocumentForURL:absoluteDocumentURL
 withContentsOfURL:absoluteDocumentContentsURL
 error:outError];
 if (success) {
 if ([[self documentForURL:absoluteDocumentContentsURL]
 isKindOfClass:[DiaryDocument class]]) {
 [[NSNotificationCenter defaultCenter] postNotificationName:
 VRDidRestoreAutosavedDiaryDocumentNotification
 object:self];
 }
 }
 return success;
}
```

This is similar to what you did in other override methods in this recipe to post a notification. The important difference is how you determine that a Chef's Diary document is the kind of document that is being restored. The `absoluteDocumentContentsURL` argument is the URL for the autosaved document, which is `~/Library/Autosave Information` if the user has never saved it, or some other location if the user has saved it. You use this URL to get the document and determine whether its class is `Diary Document`.

Declare and define the new notification string. Near the top of the `VRDocumentController.h` header file, add this declaration after the existing `VRDefaultDiaryDocumentAliasDataKey` declaration:

```
extern NSString *VRDidRestoreAutosavedDiaryDocumentNotification;
```

At the bottom of the `VRDocumentController.m` implementation file, add this definition:

```
NSString *VRDidRestoreAutosavedDiaryDocumentNotification =
 @"did restore autosaved diary document notification";
```

Register to observe the notification by inserting this statement at the end of the `-windowDidLoad` method in the `DiaryWindowController.m` implementation file:

```
[defaultCenter addObserver:self
 selector:@selector(didRestoreAutosavedDiaryDocument:)
 name:VRDidRestoreAutosavedDiaryDocumentNotification object:nil];
```

You have already taken care of removing `DiaryWindowController` as an observer, in the `-windowWillClose:` method.

10. Finally, implement the notification method that responds to the notification. In this method, you present an informational alert in the form of a sheet attached to the window of the just-restored autosaved diary document. This is a standard way to present alerts to the user, and you are likely to use it many times.

Define the notification method after `-didAutosaveDiaryDocument:`, in the Notification Methods section of the `DiaryWindowController.m` implementation file, like this:

```
- (void)didRestoreAutosavedDiaryDocument:(NSNotification *)notification {
 NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
 if (![defaults boolForKey:
 DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY]) {
 [[self alertDidRestoreAutosavedDiaryDocument]
 beginSheetModalForWindow:[self window]
 modalDelegate:self didEndSelector:
 @selector(alertDidRestoreAutosavedDiaryDocumentDidEnd:
 returnCode:contextInfo:) contextInfo:NULL];
 }
}
```

The notification method does not display the alert at all if the user has set a Boolean value in the user defaults keyed to `DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY`. You haven't written the code to let the user do that yet, but go ahead and define the macro now: At the top of the `DiaryWindowController.m` implementation file, insert this above the `@implementation` directive:

```
#pragma mark MACROS

#define DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY
 @"alert restore diary document suppressed"

#pragma mark -
```

The `pragma mark` statements set up the Xcode function menu appropriately.

There are a few different ways to present alerts, but in your ongoing work as a developer, you are likely to use `NSAlert`'s `-beginSheetModalForWindow:modalDelegate:didEndSelector:contextInfo:` method most often. It uses the same design pattern you saw in Recipe 4, where you implemented the `-saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo:` method and similar methods. You specify a callback selector and a temporary modal

delegate to handle whatever the user does in the alert after the user dismisses it. Here, as is usually the case, you designate `self` as the modal delegate. You specify the callback selector as `alertDidRestoreAutosavedDiaryDocumentDidEnd:returnValue:contextInfo:`, and you will write that method shortly.

First, create the alert. Most example code does this inline, in the same method that displays the alert. I prefer to create all my alerts in separate methods, which I gather near the end of the source files, where I can easily examine the wording and format of all my alerts at once to help ensure consistent style. I group the callback methods with them.

In the `DiaryWindowController.h` header file, just before the `@end` directive at the end of the `DiaryWindowClass` interface (before the several protocol and subclass declarations), declare this method:

```
#pragma mark ALERTS
```

```
- (NSAlert *)alertDidRestoreAutosavedDiaryDocument;
```

At the end of the `DiaryWindowController` class definition in the `DiaryWindowController.m` implementation file, define the method like this:

```
- (NSAlert *)alertDidRestoreAutosavedDiaryDocument {
 NSAlert *alert = [[[NSAlert alloc] init] autorelease];
 [alert setMessageText:[NSString stringWithFormat:
 NSLocalizedString(@"Document \"%@" was restored from an
 autosaved copy.", @"message text for
 DidRestoreAutosavedDiaryDocument alert"),
 [[self document] displayName]]];
 [alert setInformativeText:NSLocalizedString(@"Save it before
 you close it to avoid losing any changes.", @"informative
 text for DidRestoreAutosavedDiaryDocument alert")];
 [alert setShowsSuppressionButton:YES];
 return alert;
}
```

This is a standard technique for creating alerts. You allocate, initialize, and autorelease an `NSAlert`; then you set its message text, its informative text, and other features available through the `NSAlert` class, such as setting the suppression button feature you see here. Typically, you also set the names of all the buttons you want in the alert. Since this alert is purely informational, the only button it needs is an OK button, and `NSAlert` provides that by default if you specify no other buttons. At the end, you return the alert for use in `-didRestoreAutosavedDiaryDocument:` as the receiver of the `-beginSheetModalForwindow:modalDelegate:didEndSelector:contextInfo:` message.

Finally, implement the callback method. You name it `-alertDidRestoreAutosavedDiaryDocumentDidEnd:returnCode:contextInfo:`. The Class Reference for every class that declares a method using the temporary delegate callback design pattern describes the format that is required for the callback method, but you are free to name the callback method anything you like, in order to allow for multiple callback selectors for similar methods. Define the callback method in the `DiaryWindowController.m` implementation file immediately following `-alertDidRestoreAutosavedDiaryDocument`, like this:

```
- (void)alertDidRestoreAutosavedDiaryDocumentDidEnd:(NSAlert *)alert
 returnCode:(NSInteger)returnCode contextInfo:(void *)contextInfo {
 NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
 [defaults setBool:[alert suppressionButton] state]
 forKey:DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY];
 [defaults synchronize];
}
```

In most such callback methods, you would test which button the user clicked to dismiss the alert and take action accordingly. Since this informational alert has but the one OK button, you don't have to do anything with it. However, you do have to save the state of the suppression button to the user defaults if the user has selected it. That way, the next time an autosaved diary document is restored after a crash or a power outage, the informational alert will not be displayed, due to the test you wrote into the `-didRestoreAutosavedDiaryDocument:` notification method a moment ago.

You force the user default value to be written to disk immediately in order to facilitate testing. Otherwise, if you force-quit the application just after dismissing the dialog, the suppression button state might not be written to the user defaults, and the alert might appear again on your next test.

## 11. You're finished with autosaving and restoring the diary document's contents.

Before testing it, there are two preparatory steps you must take in order to obtain meaningful results: Be sure you always remove any previously autosaved property list file and autosaved diary document from `~/Library/Autosave Information` or from the folder where you previously saved the diary document, and remove the `com.quecheesoftware.Vermont-Recipes.plist` file from the user Preferences folder.

Then launch the application and create a new diary document; move it, resize it, reposition its split view divider, and type some text into it or create a new diary entry. After waiting at least 5 seconds, kill the application with the red Tasks button in Xcode. Then relaunch the application and see what happens. Test it before saving a diary document, and again after saving one. Test it without selecting the suppression button, and with the suppression button selected.



See the suppression alert in **Figure 7.1**.



**FIGURE 7.1** The alert presented when an autosaved document is restored.

## Step 8: Back Up the Diary Document

Closely allied to autosaving the diary document's contents is backing up the diary document automatically.

Autosaving a document's contents is a largely invisible process. The autosaved document is usually hidden from the user's view, and the user never deliberately opens it. The autosaved document is automatically deleted whenever the user saves the document or deletes it. It appears only if the application has crashed or been killed by accident.

By contrast, a backup of a document is visible and survives until the user deletes it or creates another backup. It is meant to be opened by the user if, for example, the user becomes unhappy with recent changes and prefers to go back to the previously saved version.

By default, document-based applications do not save document backups. However, it is very easy to make them keep a backup of the document as it existed at the time the user last saved the document. The document-saving process is typically atomic, which means that the old document on disk is kept around while a saved copy is written to disk. By default, the old file is then automatically deleted.

You can easily arrange to keep the old file in place, however. Its contents remain unchanged when the user starts editing the new document, so the user can easily return to the backup simply by opening it.

This is not a full-fledged backup system, of course, so don't throw away your Time Machine. But many users find it comforting to know that they can easily return to the last saved version of a document they are currently working on.

The diagram in the “Saving a Document” subsection of the “Message Flow in the Document Architecture” section of *Document-Based Applications Overview* explains that, near the end of the save operation, Cocoa calls `NSDocument`'s `-keepBackupFile` method. Cocoa's default implementation of this method returns `NO`. To turn on automatic document backups, all you have to do is override it and return `YES`.

Do it like this, at the end of the Override Methods section of the `DiaryDocument.m` implementation file:

```
- (BOOL)keepBackupFile {
 return YES;
}
```

The backup document has the same name as the current document with “~” appended to the end, such as *My Diary~.vrdiary*.

## Step 9: Implement the Revert to Saved Menu Item

In Step 7, you wrote code to autosave the Chef's Diary and to retrieve its previously autosaved contents after a crash or power failure. In Step 8, you wrote code to back up the document automatically so that the user can retrieve its previous contents even after changes have been saved. One similar operation remains to be coded: the Revert to Saved menu item in the application's File menu. This menu item allows the user to retrieve the last saved Chef's Diary contents, discarding changes made since the last save.

Currently, the application's Revert to Saved menu item doesn't work correctly. It is connected by default to its built-in `NSDocument` action method, `-revertDocumentToSaved:`, but if you choose `File > Revert to Saved` after saving the diary document and typing some new text, the application crashes. To compound the problem, the menu item is enabled under the wrong conditions. You have to write some code to get this right.

1. Start with the menu item validation problem. You’ve already learned how to validate menu items, but before tackling this menu item, you have to know where to validate it. In the *Document-Based Applications Overview*, under “The Roles of Key Objects in Document-Based Applications,” you learn that the document associated with the window that currently has keyboard focus receives first-responder action methods when the user saves, prints, *reverts*, and closes documents. If you select the Revert to Saved menu item in the Main-Menu nib file and look in the Menu Item Connections inspector, you see that the `revertDocumentToSaved:` action is already connected to the First Responder proxy. You must therefore implement a validation method to handle it in the `DiaryDocument` class.

The *Document-Based Applications Overview* also gives you a vital clue as to why the Revert to Saved menu item is currently validated under inappropriate circumstances. The menu item is properly disabled when no Chef’s Diary window is open, but as soon as you create a new one, the menu item is enabled. This is how the responder chain works by default. Creating a new, empty diary document makes it the application’s key window and places the diary document into the responder chain. When you open the File menu, the application finds its `-revertDocumentToSaved:` action method and enables the menu item. It does this even if the new diary document has never been saved and nothing exists for it to revert to, which is inappropriate.

To validate this menu item, implement the `-validateUserInterfaceItem:` protocol method in the `DiaryDocument.m` implementation file, following the new `-keepBackupFile` method, like this:

```
#pragma mark MENU ITEM VALIDATION

- (BOOL)validateUserInterfaceItem:
 (id <NSValidatedUserInterfaceItem>)item {
 SEL action = [item action];
 if (action == @selector(revertDocumentToSaved:)) {
 VRDocumentController *controller =
 [VRDocumentController sharedDocumentController];
 NSURL *currentDiaryURL = [controller currentDiaryURL];
 return [controller canOpenURL:currentDiaryURL]
 && [self isDocumentEdited];
 }
 return [super validateUserInterfaceItem:item];
}
```

This is standard user interface item validation, which you learned about in Recipe 4. Applications call this method automatically when a menu is opened. Here, you return YES to validate the Revert to Saved menu item if, and only if, the diary document can be opened and has unsaved edits. The first condition is tested using the `-canOpenURL:` method you wrote for `VRDocumentController` in Recipe 6. It returns YES only if a document has been saved and is not currently in the Trash. The second condition is tested using `NSDocument`'s built-in method, `-isDocumentEdited`, which is documented in the `NSDocument Class Reference` to return "YES if the receiver has changes that have not been saved, NO otherwise."

If you build and run the application now, you find that the Revert to Saved menu item is no longer enabled when a new, empty diary document is front-most. It is also not enabled if a previously saved diary document exists but is in the Trash. It would have been enabled if you had tested for the existence of a saved diary document using the `-currentDiaryURL` method without using the `-canOpenURL:` method to check whether it is in the Trash.

2. Turn now to making the Revert to Saved menu item work. This is a difficult nut to crack, because the process is not very thoroughly documented in the places you would naturally look first. By teasing implications out of the appropriate class references and resorting to the `NSDocument` header file and the Mac OS X 10.4 Tiger AppKit Release Notes, you will discover the true path.

What you already know, from the *Document-Based Applications Overview* and your examination of the `MainMenu.nib` file in Interface Builder, is that the action method triggered by the Revert to Saved menu item is `-revertDocumentToSaved:`, and that it is implemented in `NSDocument`. Looking up that action method in the *NSDocument Class Reference*, you learn that it calls `NSDocument`'s `-revertToContentsOfURL:ofType:error:`. The *NSDocument Class Reference* explains that this method discards unsaved changes and replaces the document's contents by reading a file or file package at the indicated URL. Great! That's just what you want. However, it doesn't work for the diary document, and the class reference says nothing about how this method is implemented.

It's time to begin the familiar and sometimes tedious and frustrating search for information. Searching the developer documentation in Xcode's documentation window won't get you very far unless you're lucky. Google turns up a few examples of developer frustration but no useful information. The next step is to examine the `NSDocument` header file. Here you hit pay dirt. The header is

heavily commented, and the comments for the `-revertToContentsOfURL:ofType:error:` method disclose that it invokes `-readFromURL:ofType:error:` and a bunch of other methods. In addition, the accompanying availability macro indicates that the `-revertToContentsOfURL: ofType:error:` method first became available in Mac OS X 10.4.

Turn to the *Mac OS X Developer Release Notes: Cocoa Application Framework (10.5 and Earlier)* and search for the `-revertToContentsOfURL:ofType:error:` method. There, in the Tiger AppKit release notes, you find several sections describing in great detail the inner workings of document-based applications in Tiger and newer. These comments also make clear the general outlines of the strategy that Apple expects you to follow. In general, Apple suggests that you invoke methods whose names begin with `-save...` and `-revert...`, but it advises that methods beginning with `-write...` and `-read...` are “there primarily for you to override” as needed.

Putting all this together, it appears that you might consider overriding `-readFromURL:ofType:error:`. It begins with `-read...` and is therefore, according to the release notes, a candidate for overriding. Furthermore, the comments about it in the `NSDocument` header file say that the “default implementation of this method just creates an `NSFileWrapper` and invokes `[self readFromFileWrapper: theFileWrapper ofType:typeName error:outError]`.” There may be other ways to achieve your goal, such as by overriding `-readFromData:`, but follow this path to see where it leads you.

In your override of the `-readFromURL:ofType:error:` method, you must use a substitute for the file wrapper technique it uses by default. The diary document in Vermont Recipes holds RTF data in an `NSTextStorage` object, which inherits from `NSMutableAttributedString`. Why not use one of the `NSMutableAttributedString` methods that read URLs? After all, as you already know, they can handle RTF data. Give it a try.

Scanning the *NSMutableString Class Reference*, you run across the `-readFromURL:options:documentAttributes:error:` method. It sounds promising. In particular, you note the comment that, in the case of RTF files, it appends the contents of the file it reads to the current contents of the data in the document, cautioning that “when using this method with existing content it’s best to clear the content away explicitly.” This is intriguingly reminiscent of the description of `-revertToContentsOfURL:ofType:error:`, which states that it discards all unsaved document modifications before replacing the contents with those read from disk. You’re on the right track.

Add this method to the `DiaryDocument.m` implementation file, after the `-keepBackupFile` method:

```
- (BOOL)readFromURL:(NSURL *)absoluteURL ofType:(NSString *)typeName
 error:(NSError **)outError {
 if ([self isRevertingToSavedDocument]) {
 NSMutableAttributedString *emptyString =
 [[[NSMutableAttributedString alloc] initWithString:@""]
 autorelease];
 [[self diaryDocTextStorage] setAttributedString:emptyString];
 NSError *error;
 BOOL success = [[self diaryDocTextStorage]
 readFromURL:absoluteURL
 options:nil documentAttributes:nil error:&error];
 if (!success) [self presentError:error];
 return success;
 } else {
 return [super readFromURL:absoluteURL ofType:typeName
 error:outError];
 }
}
```

The method first checks whether the application is currently reverting, using an accessor method you haven't yet written. The `-readFromURL:options:documentAttributes:error:` method is called by Cocoa in other situations, so if this is not a revert operation, you must call the superclass's implementation to allow other operations to take place.

If this is a revert operation, the method first discards the document's existing RTF contents by setting its text storage to an empty attributed string having no attributes, as recommended by the documentation just quoted. It then calls `NSMutableAttributedString`'s `-readFromURL:options:documentAttributes:error:` method. You pass `nil` for options and document attributes that you don't care about, and you handle any error using techniques described earlier. You then return whether the previously saved document's contents were successfully read.

3. Next, implement the `-isRevertingToSavedDocument` getter method and its associated instance variable and setter method. You did the same thing recently with the `-isRestoringAutosavedDocument` getter method.

In the `DiaryDocument.h` header file, after the `isRestoringAutosavedDocument` instance variable, insert this declaration:

```
BOOL isRevertingToSavedDocument;
```



Declare the accessor method after the `isRestoringAutosavedDocument` accessor methods:

- `(void)setIsRevertingToSavedDocument:(BOOL)flag;`
- `(BOOL)isRevertingToSavedDocument;`

Implement them in the `DiaryDocument.m` implementation file:

- `(void)setIsRevertingToSavedDocument:(BOOL)flag {  
    isRevertingToSavedDocument = flag;  
}`
- `(BOOL)isRevertingToSavedDocument {  
    return isRevertingToSavedDocument;  
}`

4. Finally, arrange to set the `isRevertingToSavedDocument` instance variable when the user chooses the Revert to Saved menu item, and reset it when the revert operation is done. This is most easily accomplished by overriding the `-revertToContentsOfURL:ofType:error:` method that the `-revertDocumentToSaved:` action method calls. The Tiger release notes suggested that you should invoke but not override `-save...` and `-revert...` methods, but you can always override a method like this if you immediately call its superclass's implementation and pass its variables straight through.

Add this method after the `-keepBackupFile` method in the `DiaryDocument.m` implementation file:

- `(BOOL)revertToContentsOfURL:(NSURL *)absoluteURL  
    ofType:(NSString *)typeName error:(NSError **)outError {  
    [self setIsRevertingToSavedDocument:YES];  
    BOOL success = [super revertToContentsOfURL:absoluteURL  
        ofType:typeName error:outError];  
    [self setIsRevertingToSavedDocument:NO];  
    return success;  
}`

5. You're done. To test your work, start with a clean slate by discarding any saved diary documents, discarding any autosaved documents, and discarding the Vermont Recipes preference file. Build and run the application, and open a new, empty Chef's Diary. Resize it, move it, reposition its divider, and, above all, type something in it. Then open the File menu, noticing that the Revert to Saved menu is disabled because you have not yet saved a diary document.

Choose File > Save As, and save the diary document on the desktop. Open the File menu again, noticing that the Revert to Saved menu item is still disabled because you haven't made any changes to its contents since saving it.

You don't have to close it now. Instead, type some more text into the window. The window is immediately marked dirty, and if you wait about 5 seconds, an autosaved copy of the document appears on the desktop. The window remains marked dirty.

Now choose File > Revert to Saved. You find that the menu is finally enabled because you have made unsaved changes to the saved diary document. An alert opens, asking whether you want to revert to the saved version of the document, discarding changes. Click Revert, and the contents of the window immediately revert to the contents at the time of the last user-initiated save. In addition, the window is marked clean, and the autosaved document icon on the desktop has been removed.

## Step 10: Build and Run the Application

You have built and run the application many times in this recipe to test each feature as it was finished. But it's always a good idea to do it again at the end of a recipe, to help you see the overall picture and spot inconsistencies and missing features.

Once again, start from scratch. Remove any leftover autosaved files from ~/Library/Autosave Information or any other folder where you saved the diary document. Remove the com.quecheesoftware.Vermont-Recipes.plist file from ~/Library/Preferences. And remove any saved diary document files that you saved from time to time. For good measure, empty the Trash.

Build and launch the application.

What's missing? Check out the menus systematically, left to right and top to bottom. The Preferences menu item in the application menu does not work. It isn't connected to an action method in Interface Builder, and it is disabled when you open the application menu. You'll implement Preferences in Recipe 10. The Print menu item in the File menu does not work. It has a connected action method in Interface Builder, but when you select it, you see a long and detailed error message in the Debugger Console that tells you, in a nutshell, that subclassing a certain method "is a subclass responsibility but has not been overridden." You'll implement printing

in Recipe 9. Go through the rest of the menus and their menu items, and you don't find another gap until you get to the Help menu. Choose Help > Vermont Recipes Help, and you see a dialog reporting that "Help isn't available for Vermont Recipes." You'll fix that in Recipe 11.

Remarkably, all the rest of the menu items are working. It is particularly fun to play with the Spelling and Grammar, Substitutions, and Transformations menu items at the bottom of the Edit menu. They give you a remarkable amount of power, and it cost you no effort at all to include them.

## Step 11: Save and Archive the Project

Quit the running application, close the Xcode project window, and save if asked. Discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like **Vermont Recipes 2.0.0 - Recipe 7.zip**. The working Vermont Recipes project folder remains in place, ready for Recipe 8.

## Conclusion

Despite the wealth of features that are now finished and available to the user in Vermont Recipes, there is more to be done. You anticipated some important features in the introduction to this recipe, such as support for Snow Leopard's new sudden termination technology. You also have to make sure the application works when running under Leopard as well as Snow Leopard, and on PowerPC Macs as well as Intel Macs.

There are a number of other features you should add to the application, including support for accessibility, and you should add Help tags to some of the application's controls.

You attend to these matters and others in the next recipe, Recipe 8, devoted to polishing the application. In subsequent recipes in Section 2 you will implement printing support, a Preferences window, a Help book, and AppleScript support, and you will prepare the application for deployment to its intended audience.

## DOCUMENTATION

---

Read the following documentation regarding topics covered in Recipe 7.

### **Class Reference and Protocol Documents**

*NSScreen Class Reference*

*NSDrawer Class Reference*

*NSWindow Class Reference*

*NSWindowDelegate Protocol Reference*

*NSUserDefaults Class Reference*

*NSNotification Class Reference*

*NSNotificationCenter Class Reference*

*NSSplitView Class Reference*

*NSToolbar Class Reference*

*NSApplicationDelegate Protocol Reference*

*NSDocument Class Reference*

*NSDocumentController Class Reference*

*NSAlert Class Reference*

### **General Documentation**

*Objective-C Programming Language (Categories and Extensions)*

*Coding Guidelines for Cocoa*

*Apple Human Interface Guidelines (Windows)*

*Interface Builder User Guide (Moving and Resizing Windows)*

*Window Programming Guide (Sizing and Placing Windows)*

*Toolbar Programming Topics for Cocoa (Calculating a Toolbar's Height)*

*Drawers*

*Binary Data Programming Topics for Cocoa*

*Notification Programming Topics for Cocoa*

*Document-Based Applications Overview (Autosaving in the Document Architecture)*

*Mac OS X Developer Release Notes: Cocoa Application Framework (10.5 and Earlier)*

# Polish the Application

In Recipe 7, you refined the diary document's usability by bringing its window into compliance with the requirements of the *Apple Human Interface Guidelines*. In this recipe, you will refine various features of the Vermont Recipes application at large in light of the requirements of the *HIG*, and you will also polish it up in other ways.

This recipe takes on a hodgepodge of tasks. Here's a roadmap: The recipe starts with a few additions and refinements to the application's menu bar. It then adds help tags to give the user a better sense of what the application's controls do, and it implements some accessibility features to help users with disabilities find their way around. It also implements some features new to Snow Leopard that make the application a better citizen within Mac OS X as a whole, such as support for sudden termination and use of the important new blocks feature. Also, it takes another look at the project build settings and tests the code to make sure that Vermont Recipes, as advertised, can run under Leopard as well as Snow Leopard. Maybe it will slip something else in as well.

## Step 1: Add a Save As PDF Menu Item

Launch TextEdit and open its File menu. There, just below the Save As menu item, you see a Save As PDF menu item. This is new in TextEdit 1.6 for Snow Leopard.

Normally, when you want to save a document as a PDF file, you follow a different procedure: Choose File > Print, and in the Print panel, open the PDF pop-up menu.

### HIGHLIGHTS

- Adding a Save As PDF menu item to save a document as a PDF file*
- Automatically alternating Show and Hide menu items*
- Manually toggling dynamic menu items and buttons*
- Using blocks in Snow Leopard to monitor modifier key events*
- Adding a second application target to the project*
- Using blocks in Snow Leopard for a Save panel completion handler*
- Using blocks in Snow Leopard for notifications*
- Adding help tags and accessibility features to the user interface*
- Supporting sudden termination in Snow Leopard*
- Using and internationalizing the application's display name*
- Creating application and document icons*
- Enabling an application to run under Leopard on PowerPC hardware*

The PDF menu contains several options relating to PDF, and it is customizable. In the Mac OS X Technology Overview, Apple refers to these options as *digital paper*.

The decision to implement the Save as PDF menu item in the Print panel has historical roots. The Portable Document Format (PDF) was created by Adobe in 1993 for document exchange. It quickly gained widespread support, and it became an open standard in 2008. It is based in part on PostScript, a page-description language that Adobe released in 1984. Apple used PostScript in its Apple LaserWriter printers in 1985, shortly after the Macintosh computer first saw the light of day in 1984. By all accounts, these printers and PostScript accounted for Apple's initial success in the marketplace, especially in the publishing industry. Although PostScript later evolved into Display PostScript for use on computer screens, its early ties to printing account for the placement of the PDF button in the Print panel.

The Snow Leopard version of the TextEdit sample code demonstrates how to implement a Save As PDF menu item in the File menu instead. The current version of the *HIG* contains a general admonition to avoid providing multiple Save As *Format* menu items, since that functionality is better placed in a Format pop-up menu in the Save panel in applications that support multiple formats. TextEdit, despite the *HIG*, leaves PDF out of its Format pop-up menu, which contains several other options, instead placing it separately in the File menu. Vermont Recipes supports only the RTF format for the Chef's Diary, and it therefore has no need for a separate Format menu in the Save As dialog. Putting a Save As PDF menu item in its File menu appears to comply with the *HIG*.

Because the TextEdit 1.6 sample code is available from Apple, you borrow from it here, with a few changes. One difference is that Vermont Recipes runs under Leopard as well as Snow Leopard. To make the new Save As PDF menu item work in Leopard, you therefore have to incorporate the long version of Apple's sample code and modify it for use under Leopard. The long version of the sample code is not in the TextEdit application but only in the read-me file that comes with it.

Looking at this code, you see that PDF capability is still inextricably linked with printing in Cocoa.

1. Begin by performing the ritual you have performed at the beginning of every recipe, incrementing the build version. Open the Vermont Recipes 2.0.0 folder in which you saved the project folder at the end of Recipe 7, leaving the archived Recipe 7 project folder where it is, and open the working Vermont Recipes subfolder.

Increment the Version in the Properties pane of the Vermont Recipes target's information window from 7 to 8 so that the application's version is displayed in the About window as 2.0.0 (8).



2. In the DiaryDocument.h header file, declare this action method after the Accessor Methods section:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)saveDocumentAsPDFTo:(id)sender;
```

Define it in the DiaryDocument.m implementation file:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)saveDocumentAsPDFTo:(id)sender {
 NSSavePanel *savePanel = [NSSavePanel savePanel];
 NSWindow *window =
 [[[self windowControllers] objectAtIndex:0] window];
 if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 [self printDocumentWithSettings:[NSDictionary
 dictionaryWithObjectsAndKeys:NSPrintSaveJob,
 NSPrintJobDisposition, nil] showPrintPanel:NO delegate:nil
 didPrintSelector:NULL contextInfo:NULL];
 } else {
 [savePanel setRequiredFileType:@"pdf"];
 [savePanel setCanSelectHiddenExtension:YES];
 [savePanel beginSheetForDirectory:nil file:nil
 modalForWindow:window modalDelegate:self didEndSelector:
 @selector(savePanelDidEnd:returnCode:contextInfo:)
 contextInfo:NULL];
 }
}

- (void)savePanelDidEnd:(NSSavePanel *)sheet returnCode:(int)returnCode
 contextInfo:(void *)contextInfo {
 if (returnCode == NSOKButton) {
 [self printDocumentWithSettings:[NSDictionary
 dictionaryWithObjectsAndKeys:NSPrintSaveJob,
 NSPrintJobDisposition, [sheet filename], NSPrintSavePath,
 nil] showPrintPanel:NO delegate:nil didPrintSelector:NULL
 contextInfo:NULL];
 if ([sheet isExtensionHidden]) [[NSFileManager defaultManager]
 setAttributes:[NSDictionary dictionaryWithObjectsAndKeys:
 [NSNumber numberWithInt:YES], NSFileExtensionHidden, nil]
 ofItemAtPath:[sheet filename] error:NULL];
 }
}
```

In the first branch of the `if` clause in the action method, which is executed only under Snow Leopard, you simply call `-printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:`, exactly as in the `TextEdit` sample code. You set all of the arguments except the first to `NO`, `nil`, or `NULL`, indicating among other things that you don't want to show the Print panel. That's all there is to it.

The key to why this generates a PDF file lies in the first argument. The *Mac OS X Snow Leopard Release Notes: Cocoa Application Framework* explains in the section called "New Support for 'Save As PDF...' in NSDocument Printing" that this method has new behavior in Snow Leopard. Specifically, if you include the value `NSPrintSaveJob` with the key `NSPrintJobDisposition` in the `NSDictionary` that you pass in the first argument, without including path or URL values, then `NSDocument` presents the standard Save panel inviting you to save it as a PDF file.

All the rest of the code is required only to make the Save As PDF menu item work under Leopard. In this branch, you have to set up the Save panel and open it explicitly, since the `-printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:` method is able to do that for you only under Snow Leopard. The `-beginSheetForDirectory:file:modalForWindow:modalDelegate:didEndSelector:contextInfo:` method requires you to implement a callback method on a temporary modal delegate, usually `self`. You do this by implementing `-savePanelDidEnd:returnCode:contextInfo:`, using the required signature described in the *NSSavePanel Class Reference*. This method is called only when the application is running under Leopard.

3. Add the menu item and connect it in Interface Builder. Open the `MainMenu` nib file, and then open the File menu in the menu bar design surface. Drag a Menu Item object from the Library window and drop it in the File menu below the Save As menu item, and then change its title to **Save As PDF...** (the three dots are a single ellipsis character, which you insert by typing Option-semicolon). Capitalize As in deference to the way `TextEdit` does it. Although the *HIG* says that menu items should be in title case, As is capitalized in the Save As menu item and Save As PDF echoes it. Control-drag from the new Save As PDF menu item to the First Responder proxy in the nib file's document window. In the HUD, select the `saveDocumentAsPDFTo:` action. Then save the nib file (**Figure 8.1**).



**FIGURE 8.1** The new Save As PDF menu item in the File menu.

4. You aren't done yet. Build and run the application to find out why, because it isn't explained in the `TextEdit` sample code or read-me file. Create a new Chef's Diary document and type a few characters into it. Then choose `File > Save As PDF`. It doesn't work, and in the Debugger Console you see an error message indicating that you are responsible for overriding the `-printOperationWithSettings:error:` method. It looks like you will have to get involved with Cocoa's printing API a little more deeply.

The *NSDocument Class Reference* states that the default implementation of the `-printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:` method calls the `-printOperationWithSettings:error:` method. The class reference states that the implementation of the latter method, like many methods that are declared and implemented in the Cocoa frameworks, does nothing. As the documentation notes, you must override it. The documentation advises you to add the passed-in print settings dictionary to a copy of the document's own `printInfo` dictionary. This brings the `NSJobDisposition` setting into the operation, along with the setting of the "Hide extension" checkbox in the Save panel and the path the user chose for the PDF file.

Taking your cue from `TextEdit`'s implementation, add the following pared-down method at the end of the Override Methods section of the `DiaryDocument.m` implementation file:

```
- (NSPrintOperation *)printOperationWithSettings:
 (NSDictionary *)printSettings error:(NSError **)outError {
 NSPrintInfo *tempPrintInfo = [self printInfo];
 if ([printSettings count] > 0) {
 tempPrintInfo = [[tempPrintInfo copy] autorelease];
 [[tempPrintInfo dictionary]
 addEntriesFromDictionary:printSettings];
 }
 NSPrintOperation *op = [NSPrintOperation printOperationWithView:
 [[[self windowControllers] objectAtIndex:0] keyDiaryView]
 printInfo:tempPrintInfo];
 if ((op == nil) && (outError != NULL)) {
 *outError = [NSError errorWithDomain:NSCocoaErrorDomain
 code:NSUserCancelledError userInfo:nil];
 }
 return op;
}
```

As suggested in the class reference, this method adds the incoming print settings dictionary to a copy of the document's `printInfo`, after checking that the print settings dictionary contains at least one setting. It then creates a print operation and returns it. We won't go into detail here about the intricacies of the Cocoa printing API, such as the role of a print operation. For now, it is enough to know that the `-printOperationWithView:printInfo:` method you call here enables the application to extract the contents of the key diary view within its entire bounds and convert it to PDF format.

This method gets the key diary view for the first parameter of the `-printOperationWithView:printInfo:` method from the `DiaryWindowController`, which is at index 0 of the document's array of window controllers.

5. Build and run the application, and go through the same routine you did before overriding `-printOperationWithSettings:error:`. This time, when you choose `File > Save As PDF` and click `Save` in the `Save` panel, the application successfully saves a PDF version of the Chef's Diary (**Figure 8.2**).

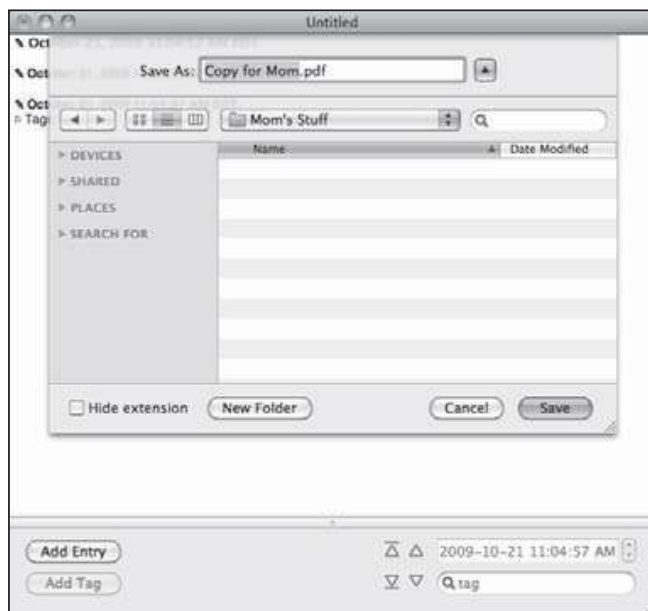


**FIGURE 8.2** The `Save As` PDF panel.

If you enter many lines of text in the Chef's Diary and then save it as a PDF file, you discover that it doesn't handle pagination very well. For example, the text on the last page is centered vertically instead of aligned to the top of the page. When you get around to doing more with the Cocoa printing API, you should consider revisiting the `-printOperationWithView:printInfo:` method to make it work well when printing the Chef's Diary to a real printer. At that time, you can deal with pagination and other formatting issues with the PDF file as well.

In Step 8, you will return to the `Save As PDF` menu item to add support for a default diary document name, *Chef's Diary*. This will entail a substantial rewrite of the `-saveDocumentAsPDFTo:` method.

## Step 2: Use Alternating Show Recipe Info and Hide Recipe Info Menu Items

In Step 5 of Recipe 5, you added a Recipe Info menu item to the application’s Window menu. Choosing it opens the recipes window’s drawer, and choosing it again closes the drawer.

The menu item does not indicate the current state of the drawer. Its title is *Recipe Info*, whether the drawer is currently open or closed. This isn’t very clear, particularly if the drawer is currently open and the recipe information is therefore already on display. It is customary for menu items that toggle the state of something in the user interface to indicate its current state, and the *Apple Human Interface Guidelines* are emphatic about advising you to avoid this kind of ambiguity, in the “Toggled Menu Items” section. For example, although I didn’t remark on it at the time, when you added a toolbar to the recipes window in Step 2 of Recipe 2, the Show Toolbar menu item that came built into the MainMenu nib file already worked, and its title automatically changes from Show Toolbar to Hide Toolbar every time you choose it. Another common technique to indicate state is to add a checkmark in front of a menu item when the state that it controls is turned on, but the Show/Hide model is more appropriate for toggling whether a user interface item is displayed.

In this recipe, you enhance the Recipe Info menu item so that it, like the Show Toolbar menu item, changes its title from Show Recipe Info to Hide Recipe Info and back again every time you click it.

1. Before beginning, give some thought to whether this menu item really belongs in the Window menu. Should it be placed in the View menu instead? The *HIG* addresses this question in two sections, “The View Menu” and “The Window Menu.” These sections tell you that the View menu should contain commands affecting “how users see a window’s content,” while the Window menu should contain commands “to select specific document windows to view or to manage a specific document window” and “to organize, select, and manage windows.” If you’re scratching your head, join the crowd.

Turn to the examples for help. They use the Finder to indicate that Show Toolbar, Show Path Bar, and Show Status Bar belong in the View menu, and the current Finder also includes Show Sidebar. These have in common the fact that each of the user interface elements is a part of a window, just as a drawer is. Furthermore, the Finder’s sidebar, like the sidebar whose Show menu item is in Preview’s View menu, is an alternative to a drawer in terms of user interface design. On balance, it seems that the Show Recipes Info menu item should be in the View menu.

Open the Main Menu nib file. In the menu bar design surface, click the Window menu's title to open the menu. Drag the Recipe Info menu item out of the Window Menu and hold it over the View menu's title. Be patient, and after a pause, the View menu opens for you. Drop the Recipe Info menu item at the top of the View menu.

Go back to the Window menu and delete one of the separators. Select it and press the Delete key. Drag a new separator menu item from the Library window and drop it below the Recipe Info menu item.

Finally, verify that the Recipe Info menu item is still connected. Select it, and then open the Menu Item Connections inspector. Sure enough, the **toggle:** action is still connected to the First Responder proxy.

2. Now change the menu item's name. Double-click the Recipe Info menu item to select it for editing, and change its title to **Show Recipe Info**.
3. While you're at it, add a keyboard shortcut similar to the Command-Option-T shortcut that Apple provides for the Show Toolbar menu item. Command-Option-R would do nicely.

Before adding a keyboard shortcut, you should always check the *HIG* to make sure it is not reserved for Apple's use and does not violate any of Apple's rules for shortcuts that are available to you. The "Keyboard Shortcuts Quick Reference" in the *HIG* contains a table listing all shortcuts that are affected by these constraints. You should also consult the table of keyboard shortcuts reserved by Universal Access features, in the "Accessibility Keyboard Shortcuts" section of *Accessibility Overview*. Finally, read *Mac OS X keyboard shortcuts*, Apple Support Article HT1343. Examining these documents, you see that Command-Option-R is available.

Select the Show Recipe Info menu item. In the Menu Item Attributes inspector, click the text field on the left in the Key Equiv. section, and then press the Command, Option, and R keys simultaneously. The symbols for the Command-Option-R shortcut appear in the text field. Press Enter or tab out of the text field to commit the new shortcut.

4. Now write some code to set the title of the menu bar item to Show Recipe Info or Hide Recipe Info, depending on the current state of the Recipe Info drawer. You learned how to do this in Recipe 6, where you created the alternating menu item titles *New Chef's Diary* and *Open Chef's Diary* in *VRDocumentController* by implementing `-validateUserInterfaceItem:`. There, you learned that the `-validateUserInterfaceItem:` method must be implemented in the target of the menu item—that is, the class that implements the action method.

The Show Recipe Info or Hide Recipe Info menu item targets the drawer, and *NSDrawer* implements the action method, `-toggle:`, that the menu item sends to that target. You must therefore implement the `-validateUserInterfaceItem:`



method in the drawer. To do this, you must subclass `NSDrawer` because you don't have access to its implementation file.

You've created several new classes already, so you know this is easy. First, keep the project window organized by creating a new group in the Groups & Files pane of the project window, using the contextual menu or choosing `Project > New Group`. Name it **Views & Responders**. I often subclass views and put them into a Views group. Views and responders have a lot in common, so putting them together is reasonable. Then select the new group, and from the contextual menu, choose `Add > New File`. Create both a header and an implementation file, and name them **RecipeInfoDrawer**. Set up both with the usual identifying information at the top.

5. Edit the `@interface` directive in the `RecipeInfoDrawer.h` header file so that the `RecipeInfoDrawer` class inherits from `NSDrawer`, and add `NSUserInterfaceValidations` in angle brackets to declare that it conforms to that protocol. The completed directive should look like this:

```
@interface RecipeInfoDrawer : NSDrawer <NSUserInterfaceValidations> {
```

6. In the `RecipeInfoDrawer.m` implementation file, define the `-validateUserInterfaceItem:` method like this:

```
#pragma mark USER INTERFACE VALIDATION
```

```
- (BOOL)validateUserInterfaceItem:
 (id <NSValidatedUserInterfaceItem>)item {
 SEL action = [item action];
 if ([item isKindOfClass:[NSMenuItem class]]
 && (action == @selector(toggle:))) {
 if ([self state] == NSDrawerClosedState) {
 [(NSMenuItem *)item setTitle:NSString(@"Show Recipe
 Info", @"menu item title for Show Recipe Info")];
 } else {
 [(NSMenuItem *)item setTitle:NSString(@"Hide Recipe
 Info", @"menu item title for Hide Recipe Info")];
 }
 }
 return YES;
}
```

You've written similar methods in Recipes 4, 6, and 7, so not much explanation is required here. You should note, however, that you test not only whether the incoming item's action is the `toggle:` action but also whether the incoming item is a menu item. Without the latter test, the method would try to set the title of the Recipe Info toolbar item, which is in the responder chain and uses the same action. The toolbar item does not implement a `-setTitle:` method, however, and the application would be unhappy with you.

7. One final requirement is that you change the class of the drawer in the RecipesWindow nib file from NSDrawer to RecipeInfoDrawer. Select the Drawer in the RecipesWindow nib file's document window, and then select the Drawer inspector and use the Class pop-up menu to change its class to RecipeInfoDrawer.
8. Build and run the application. The recipes window opens as usual. Open the View menu, and there you see the new Show Recipe Info menu item with its keyboard shortcut (**Figure 8.3**). It is enabled because the drawer is in the responder chain and implements the `-toggle:` action method. To make sure all is well, create a new Chef's Diary and open the View menu again. The Show Recipe Info menu item is disabled, along with the Hide Toolbar and Customize Toolbar menu items, because the recipes window and its drawer have been removed from the responder chain. Close the Chef's Diary or bring the recipes window to the front.

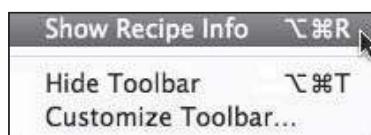


**FIGURE 8.3** The Show Recipe Info menu item in the View menu.

Choose View > Show Recipe Info. The drawer opens. Choose View > Hide Recipe Info. Yes, the name of the menu item changed to *Hide Recipe Info*, and the drawer closes (**Figure 8.4**). Use the Command-Option-R keyboard shortcut twice, and the drawer opens and closes. Out of an excess of caution, click the Recipe Info button in the toolbar twice. The drawer opens and closes.

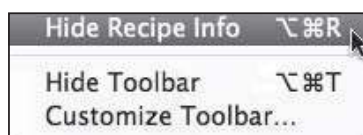


**FIGURE 8.4** The Hide Recipe Info menu item in the View menu.

## Step 3: Use a Dynamic Add Tag and Tag All Menu Item

There is another way to create alternating menu items, this time under the user's control instead of automatically in response to the changing state of the application. When the user holds down a modifier key, such as the Option key, the menu item has an alternate name and action. The *Apple Human Interface Guidelines* refers to these as *dynamic menus* in the “Naming Menu Items” section.

In this step, you enable the user to change the Add Tag menu item in the Diary menu to a Tag All menu item by holding down the Option key. You've seen behavior

like this many times. For example, in the Finder, open the File menu, and then press the Option key. While the key is down, five of the menu items change their names. For example, Close Window becomes Close All.

1. The alternate menu item and button name, Tag All, should actually do something different, of course, so first add a new action method. It scans the Chef's Diary from top to bottom looking for entries that don't already have associated tag lists, adding a tag list to all of them at once.

In the DiaryWindowController.h header file, declare the new action method after the existing `-addTag:` action method, like this:

```
- (IBAction)tagAll:(id)sender;
```

Declare it in the DiaryWindowController.m implementation file:

```
- (IBAction)tagAll:(id)sender {
 NSTextView *keyView = [self keyDiaryView];
 NSTextStorage *storage = [keyView textStorage];
 NSString *tagString = [[self document] tagTitleInsert];

 NSRange thisEntryRange = [[self document] firstEntryTitleRange];
 if (thisEntryRange.location == NSNotFound) return;
 NSRange thisTagRange;
 do {
 thisTagRange = [[self document]
 currentEntryTagRangeForIndex:thisEntryRange.location
 + thisEntryRange.length];
 if (thisTagRange.length == 0) {
 if ([keyView shouldChangeTextInRange:thisTagRange
 replacementString:tagString]) {
 [storage replaceCharactersInRange:thisTagRange
 withString:tagString];
 thisTagRange.length = [tagString length];
 [storage applyFontTraits:NSUnboldFontMask
 range:thisTagRange];
 [keyView didChangeText];
 [[keyView undoManager] setActionName:
 NSLocalizedString(@"Tag All", @"name of undo action
 for Tag All")];
 }
 }
 }
}
```

*(code continues on next page)*

```

 thisEntryRange = [[self document]
 nextEntryTitleRangeForIndex:thisEntryRange.location
 + thisEntryRange.length];
 } while (thisEntryRange.location != NSNotFound);

 [[self window] makeFirstResponder:keyView];
 [keyView scrollRangeToVisible:thisTagRange];
 [keyView setSelectedRange:NSMakeRange(thisTagRange.location
 + thisTagRange.length, 0)];
}

```

The `-tagAll:` action method is quite similar to the `-addTag:` action method. The only difference is that, instead of inserting a tag title, if needed, in the current diary entry based on the location of the insertion point, it starts at the beginning of the file and adds a tag title, if needed, in every diary entry based on a moving index.

2. Turn to Interface Builder and set up the alternate menu item that sends your new action message.

The first part of setting up the alternate menu item—adding it to the menu bar, naming it, and connecting its action—is familiar. Open the MainMenu nib file, and in the menu bar design surface, open the Diary menu. From the Library window, drag a menu item to the Diary menu and drop it below the Add Tag menu item. Double-click the new menu item and name it **Tag All**. Control-drag from the new Tag All menu item to the First Responder proxy in the MainMenu nib file's document window and select the `tagAll:` action.

Now turn it into a true alternate menu item. Select the new Tag All menu item. In the State section of the Menu Item Attributes inspector, select the Alternate checkbox. This tells Interface Builder that the Tag All menu item is an alternate to the preceding menu item, the Add Tag item. With the Tag All menu item still selected in the menu bar design surface, click the text field on the left in the Key Equiv. (for Equivalent) section of the inspector and press the Option key on the keyboard. Then press the Enter key or tab out of the text field to commit the entry. Save the nib file.

This rigmarole is documented in the *NSMenuItem Class Reference* for the `-setAlternate:` method, with sample code in case you prefer not to use Interface Builder. Cocoa treats a menu item as an alternate for another menu item if they have the same key equivalent but different modifiers. Here, both menu items have the empty string as a key equivalent. They have different modifiers; specifically, the Tag All menu item has the Option key as its modifier, and the Add Tag menu item has no modifier. As a result of these arrangements, Cocoa displays the Add Tag menu item when you hold down no modifier keys, and it displays the Tag All menu item when you hold down the Option key. When you choose one or the other, the associated action is sent.

3. Don't forget to attend to validation to enable and disable the alternate menu item. Add the following lines to the existing `-validateUserInterfaceItem:` method in the `DiaryWindowController.m` implementation file, just after the similar lines for the `addTag:` action:

```
} else if (action == @selector(tagAll:)) {
 return ([[self document] firstEntryTitleRange].location
 != NSNotFound);
}
```

If there are no entries, you can't add tags and the menu item is disabled. If there is at least one entry, you might be able to add tags. For validation purposes, don't search the entire Chef's Diary to see whether all the entries already have tags.

4. You're ready to see whether it works. Build and run the application. Open the Diary menu by clicking it, so that it stays open, and you see that all its menu items are disabled because no diary document is open yet. Press the Option key, and the alternate Add Tag menu item nevertheless changes to Tag All. So far, so good.

Create a new Chef's Diary document, but leave it empty for the moment. Open the Diary menu again. The Add Entry menu item is now enabled because you can always add more entries to the Chef's Diary, as long as it's open. The Add Tag menu item is still disabled, because you can't add tags unless there is at least one entry in the Chef's Diary (**Figure 8.5**). Press Option. The alternate Tag All menu item appears and is also disabled, for the same reason.

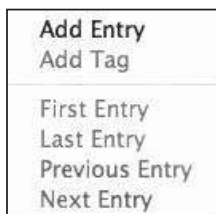


**FIGURE 8.5** The Add Tag menu item disabled in the Diary menu.

Now add a whole bunch of entries, using the Add Entry button in the diary window. Add tag lists to some of them by placing the pointer in various entries and clicking the Add Tag button for each. Now open the Diary menu again and hold down the Option key. The alternate Tag All menu item is enabled (**Figure 8.6**). Choose it, and tag lists are instantly added to all of the entries that didn't have them to start with.



**FIGURE 8.6** The alternate Tag All menu item enabled in the Diary menu.

Now for the acid test: Choose Edit > Undo Tag All. The tag lists you added using the Tag All menu item disappear instantly, leaving the tag lists you added individually in place. Choose Edit > Redo Tag All, and all the tag lists return.

## Step 4: Use a Dynamic Add Tag and Tag All Button

I don't recall seeing many applications that use alternate button titles that toggle under user control the way alternate menu items do. One example is the standard Cocoa Find panel's Save All button, which toggles to In Selection when the Option key is down. It's especially easy to do in Snow Leopard. You'll do it here so that the Add Tag button, which is the exact counterpart of the Add Tag menu item, similarly changes to a Tag All button when the user holds down the Option key. I call this a *dynamic button*.

This is not the same as the alternate button title feature supported by NSButton. The built-in alternate button title supports titles that reflect the button's on or off state.

To set up a dynamic button, use the new blocks feature, which is pervasive in Snow Leopard.

Because an application containing blocks-based code won't launch under Leopard, you create a second application target in this step for Snow Leopard only. The new target is identical to the existing application target except that its deployment target is set to Mac OS X 10.6 Snow Leopard. That is, it runs only under Snow Leopard. The old target runs under both Leopard and Snow Leopard, but even when run under Snow Leopard, it does not incorporate the dynamic button feature. When the time comes to release the application in the marketplace, you will have to take care how you identify the two versions so that your users don't become confused, but for now you care only about making both versions work as intended.

You don't have to create a duplicate set of code files to do this. Instead, you add availability macros to some of your existing code files. The compiler skips over the new blocks-based code when you build the project for the old target.

One of many new blocks-based methods in Snow Leopard is NSEvent's `+addLocalMonitorForEventsMatchingMask:handler:` class method. The signature of this method reflects two of the naming conventions associated with blocks, monitor and handler. These terms capture the concepts perfectly.

When you call this method, it installs an *event monitor* that sits behind the scenes monitoring what is happening. This is different in concept from the way alternate menu items work, at least when the user holds down a modifier key before opening the menu. Menu items normally determine their state and other features at specific times—namely, when the user opens the menu. To be sure, an alternate menu item also shows its face when the user presses the modifier key while the menu is held open. A blocks-based monitor is like the menu while it's being held open. The monitor is watching all the time to see what is happening. In this case, it monitors for a specific keyboard event, pressing or releasing the Option key.



## Blocks

---

The blocks API is new in Snow Leopard, and blocks-based methods now appear throughout the Cocoa frameworks. In some cases, the older non-block methods are marked as deprecated in Snow Leopard, so you should use the new blocks-based counterpart in your Snow Leopard applications. Apple has made blocks open source and proposed them as an addition to the C standard. The message is clear. Blocks have arrived. They're here to stay, and you have to learn to live with them.

I suspect the transition will not be easy for some. Blocks use syntax that is similar to that of C function pointers, and it's more than a little scary at first. Books on Cocoa, including this one, tend to encourage newcomers by telling you that you don't really have to know the finer points of C to learn Objective-C and the Cocoa frameworks. This is true for the most part, but it has always been the case that there are some Cocoa methods somewhere that require you to learn virtually all of the more arcane aspects of C. Now, function pointer syntax and the new block syntax have joined that group.

In Objective-C, blocks are objects that can be created and assigned to variables in your code or used as parameters to functions and methods. Blocks capture local state by copying the values of local variables in the current context, and they can mutate local state if desired. They can be passed around as a form of data and reused. The syntax for doing this—for making use of parameter values and using other operations inside a block, and for integrating blocks into the rest of your code—looks quite different from familiar Objective-C usage, although blocks are treated as ordinary Objective-C objects in Objective-C code. I won't go into detail about the syntax here. Read Apple's *Blocks Programming Topics* and the "Block Objects" section of *Introduction to Blocks and Grand Central Dispatch* to get it from the horse's mouth.

In the Objective-C context, a block is an object declared like a C function pointer but substituting a caret (^) for an asterisk (\*). In the blocks-based method you use in Step 4, for example, the declaration of the block argument to the method's handler: parameter looks like this:

```
(NSEvent* (^)(NSEvent*))block
```

This can be referred to as an anonymous block. If you prefer to give it a name, you might declare the variable like this:

```
(NSEvent* (^myHandler)(NSEvent*))block
```

The first part, `NSEvent*`, indicates that the block returns an `NSEvent` object. The next part, `(^)` or `(^myHandler)`, identifies this as an unnamed block or a block named *myHandler*. The third part, `(NSEvent*)`, indicates that the block takes a single unnamed parameter, another `NSEvent` object. If this block took additional parameters, they would be separated by commas in the parameter list.

*(continues on next page)*

The `^` operator marks the beginning of the block expression. When you use a block as a function or method argument, the code that forms the body of the block is often written inline, in curly braces.

In Objective-C, blocks can be passed around in your code, and block types can be declared as typedefs. Blocks start out on the stack, but you can move them to the heap by sending them the `-copy` message. You must balance every call to `-copy` and `-retain` with a call to `-release` for correct memory management.

Blocks are particularly useful in code that uses contextual information, such as callbacks and custom iterations. Blocks-based methods in the Cocoa frameworks often do this, and Apple has adopted naming conventions that emphasize this usage. The blocks-based method you use in Recipe 4, for example, refers to a *monitor* and a *handler*. Several new Snow Leopard methods take a parameter called a *completion handler*, which takes the place of a modal delegate callback method in the older methods. Other methods use blocks for iterating over, searching, or sorting collections such as `NSArray`s.

The developer community appears to be fascinated with blocks. A host of articles have appeared recently, only some of which are listed in the “Documentation” sidebar at the end of Recipe 8. Once you get used to them, blocks are likely to make your code simpler and some of your application’s operations much faster.

There is one awkward aspect of blocks that you will encounter in this step: Not only do they not work when running under Leopard, but if your application includes blocks-based code, the application won’t even launch under Leopard. Currently, Apple does not document this behavior. You can’t cure the problem by branching around the blocks-based code, as you have done several times to avoid calls to methods that only exist in Snow Leopard.

The reason for this is that blocks use a runtime library that isn’t available on Leopard, and it is not a weak-linked library. If the dynamic loader does not see this runtime library when you launch your application, the application’s icon bounces a few times in the Dock and then disappears. A system alert tells the user that the application unexpectedly quit, and an obscure error message appears in the console.

The standard way to run a blocks-based application under Leopard is to remove all blocks-based code from the compiled binary. To do this, you must use the so-called availability macros and compile the blocks-based code into a separate binary, and then ensure that the separate binary loads only when the application is launched under Snow Leopard. One way to do this is to build two versions of your application, one for Leopard users and one for Snow Leopard users.

*(continues on next page)*

This solution won't be easy to manage once you release your application, but there are examples currently on the market. A better solution is to place all blocks-based code in a separate bundle, include it in your application package, and arrange to load it at run time if the application is running under Snow Leopard. This is an advanced technique beyond the scope of this book, although it isn't all that hard to do. The easiest solutions, of course, are to abandon blocks or to abandon support for Leopard.

The availability macros and related techniques are described in *Technical Note TN2064: Ensuring Backwards Binary Compatibility – Weak Linking and Availability Macros on Mac OS X*.

There is an open source solution that allows you to write blocks-based code that will run under Leopard, but it won't let you call Apple's new blocks-based methods because they exist only in the Snow Leopard Cocoa frameworks. The open source blocks solution for Leopard, Plausible Blocks, can be found at <http://code.google.com/p/plblocks/>.

The `+addLocalMonitorForEventsMatchingMask:handler:` method also provides a *handler*—a block of code that responds when the monitor detects the specified event. It may be surprising to longtime users of Objective-C and the Cocoa frameworks that the handler is coded inline with the method that installs it, but this syntax quickly comes to feel very natural. It has the distinct advantage of allowing you to code the callback within the method that sets it up, instead of in a separate callback method that can get lost in your code files.

1. To lay the groundwork, you must first create an outlet for the Add Tag button and write a method to toggle its title and its action between the Add Tag and the Tag All functions. There is no need to prevent this code from being compiled into the Leopard target as well as the Snow Leopard-only target, because it contains no blocks-based code. The code will just sit there in the Leopard target, doing nothing.

Start with the outlet. In the `DiaryWindowController.h` header file, add an instance variable declaration at the end of the existing `IBOutlet` declarations, like this:

```
IBOutlet NSButton *addTagButton;
```

Also add an accessor method at the end of the outlet accessors, like this:

```
- (NSButton *)addTagButton;
```

In the `DiaryWindowController.m` implementation file, define the accessor:

```
- (NSButton *)addTagButton {
 return [[addTagButton retain] autorelease];
}
```

Connect the outlet in Interface Builder. In the `DiaryWindow` nib file, Control-drag from the File's Owner proxy to the Add Tag button, and choose `addTagButton` in the HUD. Then save the nib file.

2. Write the method to toggle the Add Tag button's title and action. In the `DiaryWindowController.h` header file, declare it at the end of the Utility Methods section:

```
- (void)toggleAddTagButtonForModifierFlags:(NSUInteger)modifierFlags;
```

In the `DiaryWindowController.m` implementation file, define it:

```
- (void)toggleAddTagButtonForModifierFlags:(NSUInteger)modifierFlags {
 if ((modifierFlags & NSDeviceIndependentModifierFlagsMask)
 == NSAlternateKeyMask) {
 [[self addTagButton] setTitle:NSLocalizedString(@"Tag All",
 @"name of undo action, menu item, and button for Tag All")];
 [[self addTagButton] setAction:@selector(tagAll:)];
 } else {
 [[self addTagButton] setTitle:NSLocalizedString(@"Add Tag",
 @"name of undo action and button for Add Tag")];
 [[self addTagButton] setAction:@selector(addTag:)];
 }
}
```

In a moment, you will call this method from two different places in your code, in both cases passing in a `modifierFlags` argument. This is an `NSUInteger` whose value indicates which modifier flags were being held down along with some other information. Here, you mask out the other information by logical ANDing it with the `NSDeviceIndependentModifierFlagsMask` mask constant, in order to obtain the device-independent bits of the mask. Then you test the result to see whether it is equal to `NSAlternateKeyMask`. This resolves to true only if the Option key was being held down and no other modifier keys were being held down at the same time. If so, the method sets the title of the Add Tag button to *Tag All* and sets its action to `tagAll:`. If not, the method sets the title of the Add Tag button to *Add Tag* and sets its action to `addTag:`.

3. You should install the event monitor as soon as the diary window opens, so install it in the existing `-windowDidLoad` method. You have to remove it when the window closes, so set it up as an instance variable with accessor methods now. You don't have to use availability macros with this code, either.

Declare it in the `DiaryWindowController.h` header file like this, at the end of the instance variables:

```
id eventMonitor;
```

Declare its accessor methods at the end of the Accessor Methods section, like this:

```
- (void)setEventMonitor:(id)eventMonitor;
- (id)eventMonitor;
```

Define the accessor methods in the `DiaryWindowController.m` implementation file like this:

```
- (void)setEventMonitor:(id)monitor {
 eventMonitor = monitor;
}

- (id)eventMonitor {
 return eventMonitor;
}
```

These accessors do not implement the standard memory management techniques that you first encountered in Recipe 3. Although the *NSEvent Class Reference* description of the `+addLocalMonitorForEventsMatchingMask: handler:` method does not say so, you normally don't have to retain and release the event monitor. This information is hidden where you should always look when nothing else gives you an answer: in the header file, in this case, `NSEvent.h`.

4. Now install the event monitor. At the end of the `-windowDidLoad` method in the `DiaryWindowController.m` implementation file, add this:

```
#if MAC_OS_X_VERSION_MIN_REQUIRED >= MAC_OS_X_VERSION_10_6
 [self setEventMonitor:[NSEvent addLocalMonitorForEventsMatchingMask:
 NSFlagsChangedMask handler:^(NSEvent *incomingEvent) {
 [self toggleAddTagButtonForModifierFlags:
 [incomingEvent modifierFlags]];
 return incomingEvent;
 }]];
#endif
```

This code uses blocks, so you enclose it in a compiler directive using the availability macro `MAC_OS_X_VERSION_MIN_REQUIRED`. As a result, this code is not included when you build the application target as it is currently configured, because you set the application target's Mac OS X deployment target (`MACOSX_DEPLOYMENT_TARGET`) build setting to Mac OS X 10.5 (10.5).

The stated condition is not met because, if you have set a target's deployment target in the build settings tab in the target's Info window, the `MAC_OS_X_VERSION_MIN_REQUIRED` macro is automatically set to match it. The built application will therefore run under Leopard, but when you run it under Snow Leopard, the dynamic button feature will not work. In a few moments, you will set up a second target for Snow Leopard and set its deployment target to Mac OS X 10.6 Snow Leopard. The application built from that target won't run under Leopard, but when it runs under Snow Leopard, the dynamic button feature will work.

If you've never seen a blocks-based method before, the argument passed into the `handler:` parameter probably looks really weird. It is a *block*. In the header, the block argument is declared like this: `(NSEvent* (^)(NSEvent*))block`. In the *NSEvent Class Reference*, it is described as the "event handler block object," which is passed the event to monitor. Here, the event to monitor is passed in as `incomingEvent`. It also returns an `NSEvent`, which in this case is the same `incomingEvent` that it received.

Two aspects of this code are important. First, the inline body of the block contains the code that is executed every time a keyboard event directed to the Vermont Recipes application is observed by the monitor. In other words, the block sticks around and keeps doing its job long after the `-windowDidLoad` method has gone away. Here, the block code extracts the modifier flags value of the incoming event that was just observed, `incomingEvent`, and passes it to the `-toggleAddTagButtonForModifierFlags:` method that you just wrote. That method changes the title and action of the Add Tag button if the modifier flags match those specified. You could have placed all the statements in the `-toggleAddTagButtonForModifierFlags:` method directly in the inline block, but here you placed them in a separate method because you need to reuse it.

Second, the block returns `incomingEvent`. If you fail to return the event object in this fashion, it will not be passed along to other code that might be expecting it, and the event will therefore not be noticed outside this method. This would be a disaster, because all user presses of any and all modifier keys would be blocked. Blocking a specific event by returning `nil` is sometimes useful, and you can even substitute a different event to do some very funky things. Normally, however, you should return the event without modification, as you do here.

The `+addLocalMonitorForEventsMatchingMask:handler:` method is one of several new Leopard and Snow Leopard methods that expand what you can do with events. In Snow Leopard, you can even monitor and respond to events in other applications, which opens up lots of possibilities. With the Leopard methods `+eventWithCGEvent:` and `+eventWithEventRef:`, you can easily work with the Core Graphics and Carbon event APIs, including Quartz Event Taps.



To see what you can do with Event Taps, download a free developer utility that I recently wrote, Event Taps Testbench, at <http://prefabsoftware.com/eventtapstestbench/>.

5. You should remove an event monitor when you're through with it. Here, the event monitor is needed throughout the life of the diary window, so remove it in the `-windowDidClose:` delegate method. Insert this method in the `DiaryWindowController.m` implementation file at the end of the Delegate Methods section:

```
- (void)windowWillClose:(NSNotification *)notification {
 if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 if ([self eventMonitor])
 [NSEvent removeMonitor:[self eventMonitor]];
 }
}
```

Again, the `NSEvent.h` header file indicates that you need not release the event monitor.

The outer `if` clause is included because the `-removeMonitor:` method was introduced in Snow Leopard. This method need not be enclosed in an availability macro because it does not contain any blocks-based code. It will never be called under Leopard.

6. Before you can test the new dynamic button, you must build a version of the application that has its deployment target set to Snow Leopard so that the blocks-based code you just wrote will be compiled into the executable. You could easily do this by changing the deployment target setting in the existing application target, but then you would lose the ability to test the application under Leopard until you change it back. Instead, create a new, identical target application in the project, change its deployment target to Snow Leopard, and build the application from it. From now on in this book, you will always build this new Snow Leopard-only target, so that you can test all of your new blocks-based code, but you will be able to build the other target any time you want to test it on Leopard.

To duplicate the existing Vermont Recipes application target, open the contextual menu on it and choose Duplicate. A new target is created and appears in the Groups & Files pane as *Vermont Recipes Copy*. Double-click to edit its name, and rename it **Vermont Recipes SL** (for Snow Leopard). Then open the new target's Info window and change the Mac OS X Deployment Target (MACOSX\_DEPLOYMENT\_TARGET) to Mac OS X 10.6 using the pop-up menu. Be sure to set it to Mac OS X 10.6 for both the Debug and the Release build.

7. Build and run the application to test the new functionality. To build the application from the new Snow Leopard target, use the Overview pop-up menu to choose Vermont Recipes SL as the active target. The active executable automatically changes to Vermont Recipes SL. Use the Overview menu again to choose the Debug configuration. Then click the Build and Run toolbar item.

Create a new Chef's Diary document, and, once again, add several entries and add tag lists to some of them (**Figure 8.7**). Now hold down the Option key. The Add Tag button immediately turns into a Tag All button (**Figure 8.8**). Click it, and tag lists are immediately added to all of the diary entries that don't already have them. Release the Option key, and the button turns back into the Add Tag button.



**FIGURE 8.7** The Add Tag button at the bottom of the Chef's Diary window.



**FIGURE 8.8** The Tag All button at the bottom of the Chef's Diary window.

But there is a problem, and if you read the *NSEvent Class Reference* or the *NSEvent.h* header file carefully, you know what it is. The monitor does not detect keyboard events while a menu is open, so your handler is not called and the Add Tag button does not turn into a Tag All button. Try it. Open the Diary menu or any other Vermont Recipes menu, and while it is open, hold down the Option key. If you're holding open the Diary menu, the Add Tag menu item changes to Tag All, but the Add Tag button does not. If you close the menu while still holding down the Option key, the button remains an Add Tag button because the button missed the Option-key-down event. This results from the way menu tracking is implemented in Mac OS X, as well as control tracking, window dragging, and various other operations.

Perhaps your users will never notice this behavior or be bothered by it, but you can fix it up to a point. You should fix it, because the Diary menu's Add Tag menu item and the Add Tag button should be coordinated as closely as possible.

A way to fix this issue is to force the Add Tag button's identity to change when the user closes any menu if the Option key is still down. Implement the `-menuDidClose:` delegate method to do this, and if the Option key is down, call the `-toggleAddTagButtonForModifierFlags:` method that you just wrote.

You can't use Interface Builder to make `DiaryWindowController` the delegate of the application's menus, because the menus are in the `MainMenu` nib file, while `DiaryWindowController` is in another nib file. You'll have to do this programmatically. At the end of the `-windowDidLoad` method in the `DiaryWindowController.m` implementation file, add this code inside the availability macro testing for Snow Leopard:

```
for (NSMenuItem *thisMenuItem in [[NSApp mainMenu] itemArray]) {
 [[thisMenuItem submenu] setDelegate:self];
}
```

This makes use of the fact that the main menu controlled by `NSApplication` contains an array of menu bar items, and each menu bar item has a submenu that is one of the main menus of the application, such as the Edit menu. By looping through all of the menu bar items and making the window controller the delegate of the submenu of each menu bar item, you accomplish the task.

To avoid a compiler warning when building this under Snow Leopard, you must declare that the `DiaryWindowController` class conforms to the `NSMenuDelegate` protocol. To do this, edit the `DiaryWindowController.h` header file so that the `@interface` declaration looks like this:

```
@interface DiaryWindowController : NSWindowController
 <NSUserInterfaceValidations, NSMenuDelegate> {
```

You should set the delegate relationships to `nil` when the window closes, to remove any risk that you might try to send a message to the delegate after the window is closed, when the window controller is no longer available. Add this code at the end of the `-windowWillClose` delegate method that you just wrote, again inside the `if` block testing for Snow Leopard:

```
for (NSMenuItem *thisMenuItem in [[NSApp mainMenu] itemArray]) {
 [[thisMenuItem submenu] setDelegate:nil];
}
```

Finally, add the `-menuDidClose` delegate method at the end of the Delegate Methods section of the `DiaryWindowController.m` implementation file:

```
- (void)menuDidClose:(NSMenu *)menu {
 if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 [self toggleAddTagButtonForModifierFlags:[NSEvent modifierFlags]];
 }
}
```

This uses a new class method in Snow Leopard, `+[NSEvent modifierFlags]`. It returns the current state of the modifier keys at the moment it executes, outside of

the event stream. You pass this value to the `-toggleAddTagButtonForModifierFlags:` when the user closes any of the application's menus. If the Option key is held down, the button turns into a Tag All button.

8. Test it again. This time, when you close any menu item while holding down the Option key, the Add Menu button immediately turns into a Tag All button. This doesn't happen while the user is holding open the menu and holding down the Option key at the same time, but it does happen after the menu closes if the user continues to hold down the Option key. That's better than leaving the user wondering why the Option key no longer seems to change the button to its alternate identity.

## Step 5: Use Blocks for Notifications

In Recipe 7, the diary document used notifications to inform the diary window controller when the user saved the document, when the application autosaved the document, and when the application restored the document after a crash or a power outage. To be precise, the diary document posted notifications of those events to the default notification center, and the diary window controller registered to observe those notifications and act on them when they arrived.

Now that you have learned a little something in this recipe about the new blocks-based methods in Snow Leopard, you should take advantage of the blocks-based notification registration method introduced in Snow Leopard, `-addObserverForName:object:queue:usingBlock:`. This does not require a wholesale rewrite of the code you wrote in Recipe 7. The code to post the notifications and the code to act on them remains unchanged. The only change is in how you register to observe them and to unregister when you're done with them.

As with many of the other blocks-based methods in Snow Leopard, using blocks in this case enables you to write the code to be executed later inline with the code that sets it up. You can still leave the code to be executed later in separate methods, as you do here, but you call those methods inline. Eventually, when your application no longer needs to support pre-Snow Leopard APIs, you can move the body of the separate methods into the inline block declaration for simplicity.

The blocks-based `-addObserverForName:object:queue:usingBlock:` method actually requires you to write a little more code than the old method. You have to keep the observer around by assigning it to an instance variable, and you use the instance variable to remove the observer later. In exchange for this additional effort, you gain a little better logical precision in removing the observer in some circumstances, and you gain improved locality and readability in your code.

1. Start with the instance variable and accessor methods. This is essentially the same as the code you wrote for the `eventMonitor` instance variable in Step 4. Declare three new instance variables after the `eventMonitor` declaration in the `DiaryWindowController.h` header file, like this:

```
id didSaveDiaryDocumentObserver;
id didAutosaveDiaryDocumentObserver;
id didRestoreAutosavedDiaryDocumentObserver;
```

Declare their accessor methods at the end of the Accessor Methods section, like this:

- (void)setDidSaveDiaryDocumentObserver:(id)observer;
- (id)didSaveDiaryDocumentObserver;
  
- (void)setDidAutosaveDiaryDocumentObserver:(id)observer;
- (id)didAutosaveDiaryDocumentObserver;
  
- (void)setDidRestoreAutosavedDiaryDocumentObserver:(id)observer;
- (id)didRestoreAutosavedDiaryDocumentObserver;

Define the accessor methods in the `DiaryWindowController.m` implementation file like this:

- (void)setDidSaveDiaryDocumentObserver:(id)observer {  
 didSaveDiaryDocumentObserver = observer;  
}
  
- (id)didSaveDiaryDocumentObserver {  
 return didSaveDiaryDocumentObserver;  
}
  
- (void)setDidAutosaveDiaryDocumentObserver:(id)observer {  
 didAutosaveDiaryDocumentObserver = observer;  
}
  
- (id)didAutosaveDiaryDocumentObserver {  
 return didAutosaveDiaryDocumentObserver;  
}
  
- (void)setDidRestoreAutosavedDiaryDocumentObserver:(id)observer {  
 didRestoreAutosavedDiaryDocumentObserver = observer;  
}
  
- (id)didRestoreAutosavedDiaryDocumentObserver {  
 return didRestoreAutosavedDiaryDocumentObserver;  
}

As with the `eventMonitor` accessor methods, these accessors do not implement the standard memory management techniques. The *NSNotificationCenter Class Reference* description of the `-addObserverForName:object:queue:usingBlock:` method claims that you do have to retain the observer, saying, “You must retain the returned value as long as you want the registration to exist in the notification center.” But the class reference is wrong. The *Mac OS X Snow Leopard Release Notes: Cocoa Foundation Framework* gets it right, in the “NSNotificationCenter new API” section. The Release Note explains that “the system maintains a retain on this object (until it is removed),” which is a way of saying that you don’t own the observer and therefore do not have to retain and release it. The Release Note explains that you nevertheless do have to “keep a reference” to the observer, meaning that you must assign it to an instance variable. This is for two reasons. One is to enable you to remove the observer when you’re done with it, which you will do in a moment. The other is to prevent the garbage collector from collecting it prematurely if you use Garbage Collection in your application. You will learn about Garbage Collection later.

2. Now register the observers. Reuse the registration code you wrote in Steps 4 and 7 of Recipe 7 because they are needed when the application is running under Leopard. The blocks-based methods were introduced in Snow Leopard. You must again use the availability macros to make sure the Snow Leopard version of the built application does not include the blocks-based code. In this case, both versions of the application will have the functionality of the notifications, but the Snow Leopard version will do it the new way.

Near the end of the `-windowDidLoad` method in the `DiaryWindowController.m` implementation file, insert this availability macro between the declaration of the `defaultCenter` local variable and the existing registration statements:

```
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_6
```

Immediately following the existing registration statements, add this new `#else` block:

```
#else
 [self setDidSaveDiaryDocumentObserver:[defaultCenter
 addObserverForName:VRDidSaveDiaryDocumentNotification
 object:nil
 queue:nil
 usingBlock:^(NSNotification *notification) {
 [self didSaveDiaryDocument:notification];
 }]];
 [self setDidAutosaveDiaryDocumentObserver:[defaultCenter
 addObserverForName:VRDidAutosaveDiaryDocumentNotification
 object:nil
```



```

 queue:nil
 usingBlock:^(NSNotification *notification) {
 [self didAutosaveDiaryDocument:notification];
 }]];
 [self setDidRestoreAutosavedDiaryDocumentObserver:[defaultCenter
 addObserverForName:
 VRDidRestoreAutosavedDiaryDocumentNotification
 object:nil
 queue:nil
 usingBlock:^(NSNotification *notification) {
 [self didRestoreAutosavedDiaryDocument:notification];
 }]];
#endif

```

These are three calls to the blocks-based observer registration method `-addObserverForName:object:queue:usingBlock:`. Each of them assigns the observer that the method returns to one of the three instance variables you just created. The name and object parameter values are the same as those you used in the old-style method calls, and they serve the same purpose. You already set up the name constants in Recipe 7. The queue parameter enables you to run the block asynchronously in an `NSOperation` object. For present purposes, synchronous execution is appropriate, so you pass `nil` in this parameter.

The block parameter value in each call consists of nothing more than a call to the corresponding notification method you wrote in Recipe 7, such as `-didSaveDiaryDocument:`, passing an `NSNotification` object to it. You could place the body of the corresponding notification method in the block, but since you need the separate notification methods for Leopard, you should leave them as separate methods.

There is one bit of cleanup required, however. In Step 4 of Recipe 7 you noted that it wasn't necessary to declare `-didSaveDiaryDocument:` and its two sister methods because you executed their selectors directly instead of sending them as messages. Here, however, in the Snow Leopard version, you do send them as messages. Therefore, in the `DiaryWindowController.h` header file, you must add these declarations after the Action Methods section:

```

#pragma mark NOTIFICATION METHODS

- (void)didSaveDiaryDocument:(NSNotification *)notification;
- (void)didAutosaveDiaryDocument:(NSNotification *)notification;
- (void)didRestoreAutosavedDiaryDocument:(NSNotification *)notification;

```

Finally, what is this mysterious observer object that the `-addObserverForName:object:queue:usingBlock:` method returns? Apple's documentation doesn't really say. The class reference describes it, in the Objective-C context, as an object that conforms to the `NSObject` protocol.

3. Finally, unregister, or remove, the observers.

In Recipe 7, you added a call to the notification center's `-removeObserver:` method to the end of the `-windowDidClose:` delegate method, passing `self`, the window controller, as the observer to be removed. This use of the `-removeObserver:` method is common. It removes the observer object no matter how many different notifications it has registered to observe. For Leopard, there is also a more specific removal method for subsets of observers, `-removeObserver:name:object:`.

Here, you also use the `-removeObserver:` method, but to remove blocks-based observers, you have to call it once for each observer, passing the observer in the parameter. This is one reason why you had to keep the observers in instance variables.

Near the end of the `-windowDidClose:` method in the `DiaryWindowController.m` implementation file, insert this immediately before the existing call to the default notification center's `-removeObserver:` method:

```
if (floor(NSFoundationVersionNumber)
 <= NSFoundationVersionNumber10_5) {
```

Then, just after the existing call to `-removeObserver:`, add this `else` branch:

```
} else {
 [defaultCenter removeObserver:
 [self didSaveDiaryDocumentObserver]];
 [defaultCenter removeObserver:
 [self didAutosaveDiaryDocumentObserver]];
 [defaultCenter removeObserver:
 [self didRestoreAutosavedDiaryDocumentObserver]];
}
```

4. Build and run the application to test it. Perform the same tests that you performed in the fifth instruction in Step 4 of Recipe 7. They should come out the same, and they do.

## Step 6: Add Help Tags

In this step, you implement a simple way to provide help to your users, relying almost solely on Interface Builder. Specifically, you add *help tags* to a number of views and controls in the application.

Help tags, sometimes referred to by developers as *tooltips*, are short bits of text that appear in a small yellow box near the pointer a little while after the user holds the pointer over a view. Not all views support help tags, and you should not provide

them for controls whose use is self-explanatory. The pause before they appear is to help keep them out of the way of a knowledgeable user who is moving right along. They also disappear after a little while, which is a good reason to keep them short.

1. Open the DiaryWindow nib file. Select the upper-left navigation button at the bottom of the window, and open the Validated Diary Button Identity inspector. In the collapsible Tool Tip pane near the bottom of the inspector, type **Go to first entry**.

This text meets all the requirements for help tags laid out in the “User Assistance” subsection of the “Using Mac OS X Technologies” section of the *Apple Human Interface Guidelines*. It uses few words; it focuses on the single task of its user interface element; it uses sentence case and ends with a period because it is a complete sentence (sentence fragments are acceptable, but they don’t end with a period); and it tells the user what using the user interface element will accomplish.

2. Select the lower-left navigation button, and in the Tool Tip pane, enter **Go to last entry**.
3. Select the upper-right navigation button and enter **Go to previous entry**.
4. Select the lower-right navigation button and enter **Go to next entry**.
5. Select the date picker and enter **Go to oldest entry after date**.
6. Select the search field and enter **Search entries by tag**.
7. Save the nib file, and then build and run the application. Pause the pointer over one of the user interface elements, and after a moment, you see the help tag appear. Leave the pointer where it is for about 10 seconds, and you see the tag disappear.
8. As explained in the *HIG*, standard Mac OS X user interface elements such as scroll bars and the buttons in window titles should not have help tags, because users are expected to know what they do. Similarly, buttons and other controls that you add and that have meaningful titles or labels don’t need help tags. For these reasons, you don’t add help tags to the standard elements in the diary window or to the Add Entry button.

But the Add Tag button requires some thought. Because of your work in Step 4, it turns into a Tag All button when the user holds down the Option key, but how is the user supposed to know that? One problem with dynamic menu items and buttons is that the user won’t find them without experimenting with the Option key or diligently reading the help book that you will write later. In the meantime, you can help out the user by explaining this in a help tag.

In this case, however, you can’t use Interface Builder because the Add Tag button is dynamic only when the application is running under Snow Leopard. You don’t want to get Leopard users’ hopes up with a help tag promising additional goodies

that they can't use. You're in luck. All views and certain other user interface elements implement a `-setToolTip:` method that you can use to add subtler help tag behavior to the application programmatically. To use this method, you normally have to declare an outlet. Fortunately, you just did that for the Add Tag button in Step 4.

The right place to add this functionality is in the `-toggleAddTagButtonForModifierFlags:` method, which is called from the `-windowDidLoad` method. Not only is it called when the user has pressed or released the Option key, but it does this only under Snow Leopard, and it is also called when a menu is closed and the Option key is down. Insert this statement at the end of the first branch of the `if` block, to run when the user presses the Option key:

```
[[self addTagButton] setToolTip:nil];
```

And put this statement at the end of the second branch, to run when the user releases the Option key:

```
[[self addTagButton] setToolTip:NSLocalizedString(@"Press Option to
tag all entries.", @"help tag text for Add Tag button under
Snow Leopard")];
```

You also have to set the help tag when the window first opens, before the user has either pressed or released the Option key. Again, this can be done only when the user is running Snow Leopard. Add this statement at the end of the `-windowDidLoad` method, being careful to place it inside the `#if` clause of the precompiler directive that tests whether the application is for Snow Leopard only:

```
[[self addTagButton] setToolTip:NSLocalizedString(@"Press Option to
tag all entries.", @"help tag text for Add Tag button under
Snow Leopard")];
```

Now a novice user of the application has one more way to learn about the Tag All feature when running Snow Leopard (**Figure 8.9**).



**FIGURE 8.9** A help tag associated with the dynamic Add Tag button.

9. You can also add help tags to menu items, although you don't see this very often because most menu items have self-explanatory titles. As long as you're in the mood to help the user learn about the Tag All feature, add the same help tag to the Add Tag menu item. Because alternate menu items, unlike alternate buttons, really are two different menu items, you can do this entirely in Interface Builder.

Open the `MainMenu.nib` file, and then open the Diary menu in its design surface. Select the Add Tag menu item, and in the Tool Tip section of the Menu

Item Identity inspector, enter **Press Option to tag all entries**. Do not set a tooltip on the alternate Tag All menu item.

Now, when the user holds the pointer over the Add Tag menu item, the same tooltip appears (**Figure 8.10**).



**FIGURE 8.10** A help tag associated with the dynamic Add Tag menu item.

10. Toolbar items can have help tags, too. You should add one to the Recipe Info toolbar item in the recipes window's toolbar. In my experience, it is not obvious that a toolbar item opens and closes a drawer or a sidebar except in those applications where the toolbar item's name describes what it does or its icon pictures a drawer or a sidebar hanging off a window, such as the Utilities toolbar item in OmniOutliner or the Show Sidebar toolbar item in Preview.

In Vermont Recipes, the Recipe Info toolbar item discloses in neither words nor graphics that it opens and closes a drawer, so put this information into a help tag. As the *HIG* makes clear, it is perfectly all right for a single help tag to explain both parts of a toggle effect.

Open the RecipesWindow nib file and double-click the Recipes Info toolbar item. The Allowed Toolbar Items sheet opens. In it, click the Recipe Info toolbar item to select it. In the Tool Tip section of the Toolbar Item Identity inspector, enter **Open or close the Recipe Info drawer**. Save the nib file.

When you build and run the application and pause the pointer over the Recipe Info button, the help tag appears (**Figure 8.11**).



**FIGURE 8.11** A help tag associated with the Recipe Info toolbar item.

## Step 7: Add Accessibility Features

Apple introduced its accessibility technology in Mac OS X 10.2 Jaguar. It grew out of Section 508 of the federal Workforce Investment Act of 1998 and its requirements regarding access to information technology for persons with disabilities. Compliance with Section 508 is a prerequisite for sale of computer and other products to the federal

government, and to many state agencies and educational institutions. Accessibility is built into every standard Mac OS X user interface element, whether written using the Carbon or Cocoa frameworks. As long as your application uses built-in Cocoa user interface elements, there is relatively little you need to do to make it fully accessible.

For custom controls, you must provide accessibility support programmatically using the `NSAccessibility` informal protocol, as described in Apple's *Accessibility Programming Guidelines for Cocoa*. This is called *access enabling* or, colloquially, *accessorizing* a custom control. Doing this is beyond the scope of this book. Nevertheless, for the standard user interface elements in Vermont Recipes, there are a few things you should do in Interface Builder and, to some extent, in code.

Some accessibility attributes are application specific and therefore can't be built into Cocoa's standard user interface elements. Interface Builder provides support for four of these: a user interface element's description, its help text, links between elements, and element titles. If you supply text for the first two or connect the last two, this information may be recited aloud to a user who has turned on VoiceOver in the Seeing tab of the Universal Access pane in System Preferences.

The accessibility help field allows you to provide somewhat more detailed audible help tags for VoiceOver. For example, the help tag you added to one of the navigation buttons in Step 6 was "Go to first entry." This text does not mention that the pointer is in the diary window because its location is visible. In the accessibility help field, you might enter "Go to first diary entry," adding a reference to the diary to make this clear for a user who can't rely on visual information. The more detailed VoiceOver help text is spoken, instead of the less detailed help tag, when the "When an item has a help tag" pop-up menu is set to Speak Help Tag in the Hints tab of the Verbosity pane in VoiceOver Utility. Accessibility help text is spoken only for user interface elements that have a help tag.

When the "Speak instructions for using the item in the VoiceOver cursor" checkbox is selected as well, VoiceOver combines the help information with other information about how to use the user interface element and speaks these instructions aloud, as well as displaying them onscreen in the caption panel. VoiceOver usually speaks these instructions, including the help tag or the accessibility help text, only after a pause, just as help tags appear onscreen only after a pause. This does not provide the instantaneous feedback that a practiced VoiceOver user wants. For that, you must set the accessibility description field.

The rules about when to provide a description and how to word it are a bit arcane, and it often takes some trial and error to get it right. The rules are laid out in the "Access Enabling a Cocoa Application" section of the *Accessibility Programming Guidelines for Cocoa*. Perhaps the most important thing to remember is to leave out the type of the user interface element, because VoiceOver automatically adds the



type to the description. For example, in a moment you will enter *first entry* as the description of the navigation button that takes you to the first entry. If instead you entered *first entry button*, VoiceOver would speak *first entry button button*, when what you want it to speak is *first entry button*. It is essential to audit your application's VoiceOver behavior before releasing it.

You don't need to supply a description at all if the title attribute of a user interface element exists and is unambiguous, because VoiceOver ordinarily speaks the title immediately, along with the status and the generic type of the element. For example, when the VoiceOver cursor is on the Add Entry button, VoiceOver speaks "Add Entry button," and when it is on the Add Tag button, it might speak "Add Tag dimmed button." Always supply a description if an element has no title attribute. Good examples are an untitled palette and a button that has an image or icon instead of text. Because providing descriptions for images is so important, Snow Leopard adds a new `-setAccessibilityDescription:` method to `NSImage`. In addition, Snow Leopard adds strings file support, letting you place all of your application's image descriptions and their localizations in a file named `Accessibility-ImageDescriptions.strings` instead of entering descriptions separately for each image.

In Snow Leopard, you can also provide descriptions for menus and menu items that are graphical rather than textual.

Elements that don't have titles of their own are often associated on the screen with a separate label, typically a static text field. Sighted users recognize that the adjacent label is the title of the untitled element or group of elements. You should define these labels as the title for accessibility purposes, so that users with visual disabilities can receive the same information through VoiceOver. You can do this in Interface Builder or, as with all of these accessibility features, by adding a little code to your application. In code, but not in Interface Builder, you can also link the title element back to the interface element it titles by adding the `NSAccessibilityServesAsTitleForUIElementsAttribute` attribute to the title element, pointing to all of its associated untitled elements, but this return connection is rarely made in practice.

One other attribute may be supplied in Interface Builder: the link attribute. This allows you to link a user interface element in one part of a window—say, an element in the source list in a pane on the left—to all of the elements that are associated with it—say, the elements that appear in the detail pane on the right. Again, sighted users see this relationship, but you must take affirmative steps to make it accessible.

To add the attributes mentioned here to user interface elements in your application programmatically instead of by using Interface Builder, use the `-accessibilitySetOverrideValue:forAttribute:` method declared in the AppKit's `NSAccessibility` informal protocol. This way, you avoid having to subclass individual user interface elements. This method can be used only for attributes that are not user-settable, and

its use should be restricted to the four layout-based attributes described here. You must take care to use the method with the object that actually represents the user interface element in the accessibility hierarchy. For example, in the case of a button, you use the underlying `NSButtonCell`, not the `NSButton`. The accessibility hierarchy differs from the familiar view hierarchy in that accessibility ignores some elements because they are not visible to the user.

There are several tools available to help you understand the accessibility hierarchy of your application and to assess other aspects of its compliance with accessibility requirements. Two are Apple's free Accessibility Inspector and Accessibility Verifier utilities, which are installed when you install the developer tools. Another is my own UI Browser, a commercial product from Prefab Software. A 30-day free trial version of UI Browser can be downloaded at <http://prefabsoftware.com/uibrowser>.

To test the accessibility description and help attributes discussed here, turn on VoiceOver in the Seeing tab of the Universal Access pane in System Preferences. There are a lot of configuration settings available. You should go to the Verbosity pane of the VoiceOver Utility and, in its General tab, set Default Verbosity to High. This setting is the default, and when it is set, VoiceOver speaks the name, status, and type of most user interface elements when the VoiceOver cursor encounters them. You can use the VoiceOver keys, Control and Option—they are usually referred to as the VO keys—in combination with the arrow keys to move the VoiceOver cursor from element to element. However, a sighted tester may find it easier during testing to go to the Navigation pane in VoiceOver Utility and set the “Mouse cursor” pop-up menu to “Moves VoiceOver cursor” in order to use the pointer to test the buttons. A user interface element is said to be “in the VoiceOver cursor” when the cursor, a rectangular outline, encloses the element. The best way to listen is to move the VoiceOver cursor to a user interface element and then press VO-F3 (that is, the Control and Option keys with the F3 function key) for the description, VO-Shift-H for the help tag, or VO-Shift-N for instructions. This information is both spoken and displayed onscreen (**Figure 8.12**).



You are currently on a button. To click this button, press Control-Option-Space. The help tag is: Add diary tag. Press Option to tag all entries.

**FIGURE 8.12** VoiceOver display of instructions for use of the Add Tag button.

Because the Option key is one of the VO keys, it makes a difference whether you press the VO keys simultaneously or press Option first and then press Control. Pressing them simultaneously, with Shift and F3, H, or N, speaks the Add Tag button values, while pressing the Option key first followed a moment later by the other keys causes the Add Tag button to turn into a Tag All button and then speaks the Tag All button values.

It's time to put all this theory into practice.

1. The most obvious candidates for accessibility support in Vermont Recipes are the navigation buttons at the bottom of the Chef's Diary window. They are graphical buttons without titles. At present, when you move the VoiceOver cursor over one of these buttons, it speaks "dimmed button." If you add a diary entry so as to enable the two buttons that go to the first and last entries, it speaks only "button." This is not helpful.

In Interface Builder, select the navigation button that takes the user to the first entry. In the Help field in the Accessibility Identity section of the Validated Diary Button Identity inspector, enter **Go to first diary entry.** (with the period). In the description field, enter **first entry** (without the period).

2. Select the navigation button that takes the user to the last entry. In the Help field in the Accessibility Identity section of the Validated Diary Button Identity inspector, enter **Go to last diary entry.** (with the period). In the description field, enter **last entry** (without the period).
3. Select the navigation button that takes the user to the previous entry. In the Help field in the Accessibility Identity section of the Validated Diary Button Identity inspector, enter **Go to previous diary entry.** (with the period). In the description field, enter **previous entry** (without the period).
4. Select the navigation button that takes the user to the next entry. In the Help field in the Accessibility Identity section of the Validated Diary Button Identity inspector, enter **Go to next diary entry.** (with the period). In the description field, enter **next entry** (without the period).
5. Select the date picker. In the Help field in the Accessibility Identity section of the Validated Diary Date Picker Identity inspector, enter **Go to oldest diary entry after date.** (with the period). In the description field, enter **entry date** (without the period).
6. Select the search field. In the Help field in the Accessibility Identity section of the Validated Diary Search Field Identity inspector, enter **Search diary entries by tag.** (with the period). In the description field, enter **tag search** (without the period).
7. Select the Add Entry button and enter **Add diary entry.** (with the period) in the accessibility Help field. You didn't supply a help tag for this button because its title is self-explanatory to sighted users, but in the VoiceOver context it may be useful to explain that this button adds *diary* entries.

However, as noted earlier, accessibility help is not spoken unless the user interface element also has a help tag. You did not supply a help tag for the Add Entry button because the button's title is fully descriptive for sighted users. In order to provide a better experience for VoiceOver users, you might want to add a help tag for sighted users now. In the Tool Tip field, enter **Add entry.** (with the period).

Don't enter anything in the Description field, because the Add Entry button has a textual title.

8. You shouldn't use Interface Builder to add accessibility information for the Add Tag button for the same reason you didn't use Interface Builder to insert a help tag for it: The Add Tag button turns into a Tag All button when the user presses Option, but this feature is not available when the application is running under Leopard. Here, then, is your first opportunity to write some accessibility code. All of the new code goes in the `-toggleAddTagButtonForModifierFlags:` and the `-windowDidLoad` methods in the `DiaryWindowController.m` implementation file. It sets the help tag to *Tag all entries*, and it sets the accessibility help to *Tag all diary entries*, when the Option key is pressed (with trailing periods in both cases), and it sets them to *Add tag* and *Add diary tag* when the Option key is released and also when the window first loads.

Start by editing the `-toggleAddTagButtonForModifierFlags:` method. At the end of the first branch of the `if` clause, remove the existing `-setToolTip:` statement setting the tooltip to `nil` when the Option key is pressed and replace it with these two statements:

```
[[self addTagButton] setToolTip:NSLocalizedString(@"Tag all entries.",
 @"help tag text for Tag All button")];
[[[self addTagButton] cell] accessibilitySetOverrideValue:
 NSLocalizedString(@"Tag all diary entries.",
 @"accessibility tag text for Tag All button")
 forAttribute:NSAccessibilityHelpAttribute];
```

The first replacement statement simply sets the help tag's value so that it explains what the Tag All button does, just as the Add Entry button now has a help tag. The second statement is more interesting. It uses the `NSAccessibility` informal protocol's `-accessibilitySetOverrideValue: forAttribute:` method to set the help attribute of the Tag All button to provide slightly more detailed help text for VoiceOver, including the word *diary*. It sends this message to the Add Tag button's cell, not to the button itself.

At the end of the second branch of the `if` clause, remove the existing `-setToolTip:` statement and substitute these two statements:

```
[[self addTagButton] setToolTip:NSLocalizedString(@"Add tag. Press
 Option to tag all entries.", @"help tag text for Add Tag button
 under Snow Leopard")];
[[[self addTagButton] cell] accessibilitySetOverrideValue:
 NSLocalizedString(@"Add diary tag. Press Option to tag all
 entries.", @"accessibility help text for Add Tag button under
 Snow Leopard") forAttribute:NSAccessibilityHelpAttribute];
```

The first statement adds a description of what the Add Tag button does to the existing notice that the user can press Option to tag all entries. The second statement does the same thing but adds the word *diary* for more detail.

Add the same two statements to the end of the `-windowDidLoad` method, replacing the existing statement that set up the help tag to advise the user of the Tag All option. You must make one additional change to the `-windowDidLoad` method. Currently, it sets up features for use only when running under Snow Leopard. However, now that you've decided to add help tags to the Add Entry and Add Tag buttons, you have to programmatically install the Add Tag button's help tag for use while running under Leopard. Add an `#else` branch to the existing `#if` test, and insert this statement into the `#else` branch:

```
[[self addTagButton] setToolTip:NSLocalizedString(@"Add tag.",
 @"help tag text for Add Tag button under Leopard")];
```

9. The current design of the diary window doesn't provide an opportunity to use the accessibility title attribute. Perhaps it should, though, because the intended use of the date picker is not obvious even to sighted users.

Make room to add titles for the date picker and the search field by moving the four navigation buttons to the left, next to the Add Entry and Add Tag buttons. Using these two buttons has a direct impact on which of the navigation buttons are enabled, so it makes sense to associate them visually.

In the Objects tab of the Library window, locate the Label text field and drag two of them into the area of the diary window's design surface that was just vacated by the navigation buttons. In the label adjacent to the date picker, enter **Current Entry Date:** (with the trailing colon). In the other label, enter **Find Tags:** (with the colon). Then line them up with the help of the guides.

Don't forget to take care of the moved navigation buttons' and the new labels' autosizing behavior. In the Validated Diary Button Size inspector, change each navigation button so that the right strut is disabled and the left strut is enabled. In the two labels, disable the top and left struts, and enable the bottom and right struts.

Before testing these changes, temporarily add borders to the four navigation buttons by selecting the Bordered checkbox in the Validated Diary Button Attributes inspector. Otherwise, these buttons will be invisible in the Cocoa Simulator.

Choose File > Simulate Interface and try resizing the diary window. You see that the changes are working correctly, except for one thing: The navigation buttons and the new labels overlap when you make the window very narrow. To fix this problem, quit the Cocoa Simulator, select the window by clicking its title bar in the design surface, and in the Window Size inspector, enter **550** in the Width



field in the Minimum Size section. Run the Cocoa Simulator again, and that appears to be about right. Quit the simulator and remove the borders from the navigation buttons.

Now you're ready to connect the title attributes of the date picker and the search field. First, delete any descriptions in their identity inspectors. Then Control-drag from the date picker to the Current Entry Date label (**Figure 8.13**). When you release the mouse button, choose the Accessibility title attribute in the HUD (**Figure 8.14**). Accessibility now knows that the label is the date picker's title. Do the same thing to connect the search field to its label.



**FIGURE 8.13** Connecting the date picker to its label.

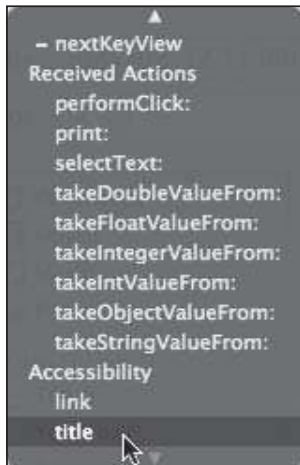


**FIGURE 8.14** Setting the date picker's Accessibility title attribute.

Save the nib file, and build and run the application to test the new title attributes. Create a new diary document, and turn on VoiceOver in the Seeing tab of the Universal Access pane of System Preferences. Then move the pointer over the search field in the diary window. VoiceOver speaks and displays *tag Find Tags: search text field*. The title attribute you just connected is working.

Now move the pointer over the date picker. This time, VoiceOver does *not* speak the title attribute. This is by design, presumably because a date picker's content is complex. Listening to the content and title together might be overwhelming, and it would take a lot of time. Leave the title attribute connected in case Apple decides to add support for a date picker's title attribute in the future.

10. Stop here. I'm not a great believer in help tags for menu items, and the Diary menu's commands are all textual in any event.

Now is therefore a good time to test your handiwork. Save the nib file, and build and run the application. Turn on VoiceOver in System Preferences (or press



Command-F5). Create a new Chef's Diary document or open an existing one. Then move the pointer over the various user interface elements at the bottom of the diary window. As you do so, VoiceOver speaks the accessibility information you provided. Pause the pointer over an element for a while, and see and hear the accessibility instructions that VoiceOver provides. Press VO-Shift-F3, VO-Shift-H, and VO-Shift-N, and note the different information provided by each. These are keyboard shortcuts commonly used by your users with disabilities, and you want to be sure you got them right.

Always audit your application's performance in VoiceOver in this manner. As an example of what can come out wrong, consider how VoiceOver would have handled the tag search field if you had not added a title attribute but instead left the accessibility description attribute set. When you moved the pointer over the search field, VoiceOver would have announced *tag tag search search text field*. That's doubly redundant. VoiceOver would first speak the word *tag* because that is the placeholder text you put in the search field. It would then speak the accessibility description you provided, *tag search*. Finally, it would speak the type of the user interface item, *search field*. You would have been well advised to solve this problem by removing the accessibility description in Interface Builder even if you had not added a title attribute.

Finally, test the Add Tag button and its alternate, Tag All. If you follow the earlier directions about pressing the VO keys simultaneously or pressing the Option key before pressing the Control key, you find that this works perfectly.

## Step 8: Provide a Default Diary Document Name

Until now, the application has encouraged the user to save a new diary document under any name whatever. The Save panel provides no guidance. It opens with the default name *Untitled* in the Save As field when the user chooses Save or Save As from the File menu. This default behavior is, of course, in conformity with the *HIG*.

However, the diary document is a one-of-a-kind document. In Step 2 of Recipe 6, you implemented the concept of a *current* Chef's Diary, only one of which can exist. While allowing the user to save it under any name, such as *Bill's Misadventures in the Kitchen*, is respectful of the user's freedom of choice, many users may prefer the convenience of a suggestion that is specific and appropriate in light of the nature of the document. In this step, you provide the default name *Chef's Diary* in the Save panel, while still allowing the user to enter any other name. This works when the application is running under Snow Leopard, but not under Leopard.

To do this, you override `NSDocument`'s `-prepareSavePanel:` method in `DiaryDocument`. This is explained in *Document-Based Applications Overview*, the document that you have consulted several times already to ascertain the chain of events that leads up to creating, saving, or opening a document. The "Saving a Document" subsection of the "Message Flow in the Document Architecture" section explains that this method is called early in the save process, just before the Save panel is opened, "to give subclasses an opportunity to customize the Save panel." This is in fact the standard technique for customizing the Save panel in any document-based application—for example, if you want to add an accessory panel to the standard Save panel.

In this case, all you want to do is to provide a default document name. Surprisingly, you couldn't do this in Leopard. The necessary method, `-setNameFieldStringValue:`, was introduced in Snow Leopard.

1. Add the following method to the `DiaryDocument.m` implementation file, just above the `-saveToURL:ofType:forSaveOperation:error:` method. You haven't been too careful about the order of the methods you've added to the Override Methods section of the file, but in this case it seems appropriate to echo the order in which the methods are called in document-based applications. Cocoa calls the `-saveToURL:ofType:forSaveOperation:error:` method shortly after calling `-prepareSavePanel:`.

```
- (BOOL)prepareSavePanel:(NSSavePanel *)savePanel {
 if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 VRDocumentController *controller =
 [VRDocumentController sharedDocumentController];
 if (![controller canOpenURL:[controller currentDiaryURL]]) {
 [savePanel setNameFieldStringValue:
 NSLocalizedString(@"Chef's Diary",
 @"default name of Chef's Diary document")];
 }
 }
 return YES;
}
```

After testing that it is running under Snow Leopard or newer, the method tests whether the user is saving a document that has not previously been saved. It does this the same way you have done it several times already, by calling `VRDocumentController`'s `-canOpenURL:` method with the value returned by its `-currentDiaryURL` method. If the diary document has never been saved or it has been saved but is currently in the Trash, the expression resolves to false. In that case, the method sets the Save As field in the Save panel to *Chef's Diary*. This method should never fail, so it returns YES.

2. Build and run the application to test what you've written.

If there is a current diary document, choosing File > Open Chef's Diary opens it, and choosing File > Save As presents a Save panel suggesting the name of the current diary document, as it should. If the "Hide extension" checkbox is deselected, the suggested name is selected so that the user can immediately begin typing to edit it, but the file extension is not selected, because it shouldn't be changed.

Close the current diary document and move its icon to the Trash. Now you can choose File > New Chef's Diary. When you choose File > Save or File > Save As, the Save panel suggests *Chef's Diary* as its name. If the "Hide extension" checkbox is deselected, *Chef's Diary* is selected so that the user can immediately begin typing to edit it. The file extension is not selected.

3. Perform one more test, and you'll see that you have a little more work to do. The Save As PDF menu item that you added in Step 1 does not exhibit the new behavior you just added to the Save As menu item. To be sure, if you save the Chef's Diary in its default RTF format under the name *Chef's Diary* or any other name, choosing File > Save As PDF suggests the saved name with the .pdf file extension. This is entirely appropriate. However, if you now close the current diary document and move it to the Trash, create a new, empty diary document, and choose File > Save As PDF, the Save panel offers to save it under the name *Untitled.pdf*.

To make the Save As PDF menu item suggest *Chef's Diary.pdf* as the file's name, you must abandon the new Snow Leopard technique you used in Step 1 in the Snow Leopard branch of the `-saveDocumentAsPDFTo:` action method. That technique is a convenient shortcut to save a PDF file when you don't need to customize the Save panel. Here, however, where you do need to customize the Save panel, you must code it directly. You use a variant of the code reproduced in the TextEdit sample code's read-me file, which you also borrowed for the pre-Snow Leopard branch of the `-saveDocumentAsPDFTo:` action method. Here, you use it with a blocks-based method that is new in Snow Leopard, so you still need separate Leopard and Snow Leopard implementations.

In the `DiaryDocument.m` implementation file, delete the previous version of the `-saveDocumentAsPDFTo:` action method, and replace it with this substantially reworked version:

```
- (IBAction)saveDocumentAsPDFTo:(id)sender {
 NSSavePanel *savePanel = [NSSavePanel savePanel];
 NSWindow *window =
 [[[self windowControllers] objectAtIndex:0] window];
```

*(code continues on next page)*

```

VRDocumentController *controller =
 [VRDocumentController sharedDocumentController];
NSString *defaultPDFFileName = NSLocalizedString(@"Chef's Diary",
 @"default name of Chef's Diary document");
#if MAC_OS_X_VERSION_MIN_REQUIRED >= MAC_OS_X_VERSION_10_6
 if ([controller canOpenURL:[controller currentDiaryURL]]) {
 defaultPDFFileName = [[[controller currentDiaryURL]
 lastPathComponent] stringByDeletingPathExtension];
 }
 if (defaultPDFFileName) [savePanel
 setNameFieldStringValue:defaultPDFFileName];
 [savePanel setAllowedFileTypes:[NSArray arrayWithObject:@"pdf"]];
 [savePanel setCanSelectHiddenExtension:YES];
 [savePanel beginSheetModalForWindow:window
 completionHandler:^(NSInteger result) {
 if (result == NSFileHandlingPanelOKButton) {
 [self printDocumentWithSettings:[NSDictionary
 dictionaryWithObjectsAndKeys:NSPrintSaveJob,
 NSPrintJobDisposition, [savePanel URL],
 NSPrintJobSavingURL, nil] showPrintPanel:NO
 delegate:nil didPrintSelector:NULL
 contextInfo:NULL];
 if ([savePanel isExtensionHidden])
 [[NSFileManager defaultManager]
 setAttributes:[NSDictionary
 dictionaryWithObjectsAndKeys:[NSNumber
 numberWithBool:YES], NSFileExtensionHidden, nil]
 ofItemAtPath:[savePanel URL] path] error:NULL];
 }
 }];
#else
 if ([controller canOpenURL:[controller currentDiaryURL]]) {
 defaultPDFFileName = [[[[controller currentDiaryURL] path]
 lastPathComponent] stringByDeletingPathExtension];
 }
 [savePanel setRequiredFileType:@"pdf"];
 [savePanel setCanSelectHiddenExtension:YES];
 [savePanel beginSheetForDirectory:nil file:defaultPDFFileName
 modalForWindow:window modalDelegate:self didEndSelector:
 @selector(savePanelDidEnd:returnCode:contextInfo:)
 contextInfo:NULL];
#endif
}

```

For Snow Leopard, this uses the technique disclosed in the long version of the code in the read-me file that comes with the TextEdit sample code, except for these four differences:

First, it tests whether the file has previously been saved and is not in the Trash. If this is false, it sets the local variable `defaultPDFFileName` to *Chef's Diary*; otherwise, it sets the local variable to the name of the saved file without its file extension.

Second, it calls `-setAllowedFileTypes:` because `-setRequiredFileType:` is deprecated in Snow Leopard.

Third, it uses a new Snow Leopard key and method because the Leopard versions are deprecated under Snow Leopard. It uses the `NSPrintJobSavingURL` key instead of `NSPrintSavePath`. It also calls `-[NSSavePanel URL]` instead of `-[NSSavePanel filename]`. It does still call `NSFileManager's -setAttributes:ofItemAtPath:error:` because that method has not been given a URL-based counterpart in Snow Leopard.

Finally, it uses another Snow Leopard blocks-based method, `-beginSheetModalForWindow:completionHandler:`. The *completion handler* pattern in blocks-based methods, as explained earlier, allows you to write callback statements for later execution inline, where they are clearly associated with the method that installed them. To understand what this means, compare this blocks-based statement with the way you perform the same task in the `#else` branch. There, you declare a separate temporary delegate and in it write a separate callback method, `-savePanelDidEnd:returnCode:contextInfo:`. That callback method is executed only when the Leopard version of Vermont Recipes is running. Under Snow Leopard, the block takes its place.

The Leopard version of the action method also displays *Chef's Diary* as a suggested name for the PDF file. The `-setNameFieldStringValue:` method is not available in Leopard, but you don't need it here because you call `-beginSheetForDirectory:file:modalForWindow:modalDelegate: didEndSelector:contextInfo:` directly. Its `file:` parameter allows you to set the suggested filename.

4. Build and run the application; create a new, empty diary document; and choose File > Save As PDF. This time, the Save panel suggests *Chef's Diary.pdf* as the saved file's name. Later, when you test this while running under Leopard, you will find that it works there as well.

## Step 9: Add Support for Sudden Termination

Apple is constantly on the lookout for ways to improve system-wide performance. To this end, it has added a feature to Snow Leopard called *sudden termination*. If this brings up visions of spy thrillers and murder mysteries, it should. The idea is that the overall user experience will be improved if, when the user quits an application or logs out, the application drops dead on the spot. Some applications take a very long time to put their affairs in order, and a quick *coup de grâce* would be a blessing.

Sudden termination speeds things up by literally killing the application's process, bypassing all of the operations normally undertaken to free memory: write accumulated user defaults to disk, save unsaved document changes, and so on. Apple knows that some applications really do need time, so it provides a means by which they can cheat the undertaker until they're ready. By default, this feature is turned off. This gives you an easy out if you don't want to spend your time now to save the user's time later. But if you take your responsibilities seriously, you will see this as an opportunity to audit your application's termination behavior, to move up everything that can be moved so that it doesn't get in the way at the end, and to inform the system that your application is ready to meet its maker.

The sudden termination mechanism is very simple. To take advantage of it, you add the `NSSupportsSuddenTermination` key to the application's Info.plist and set its value to `YES`. This acts like a living will, informing the system that the application is prepared to donate its resources to applications that have a better use for them when the grim reaper comes knocking. Alternatively, you can set the key programmatically at launch. If the key is still set when the application is told to quit, the application is killed and its resources are distributed to the survivors. While the application is running, it can change the key's value from time to time to indicate that important matters require attention before it can quit, and the system will grant a reprieve. Just be sure to reset the key when you've wrapped up the loose ends.

It isn't accurate to speak of setting and resetting the key's value, as we did for convenience in the previous paragraph. The mechanism is actually based on a counter. You use two methods, `-disableSuddenTermination` and `-enableSuddenTermination`, to increment and decrement the counter. The counter is set to 1 at launch. If you set the Info.plist key to `YES` or call `-enableSuddenTermination` at launch, the counter is decremented to 0, which indicates that the application is ready to be terminated suddenly. During the life of the application, you call these methods, ideally in balanced pairs, so that the counter returns to 0 as soon as critical operations are completed. Pairs of these calls can be nested.

`NSUserDefaults` and `NSDocument` do the right thing out of the box. The former automatically disables sudden termination when changes are made to the user



defaults and automatically re-enables it when the changes are written to disk. Similarly, the latter disables sudden termination when the user makes changes to the document's data and re-enables it when the data is written to disk. In addition, promising to provide data to a pasteboard lazily disables sudden termination until all of the promised data is provided or ownership of the pasteboard changes. In an application like Vermont Recipes, therefore, you should set the Info.plist key to YES, and then scan your application's flow of execution looking for places where it would be prudent to disable sudden termination temporarily, not already covered by NSUserDefaults, NSDocument, and other built-in routines.

If you enable sudden termination, be careful what you do in any override of `-[NSDocument close]` or the `-applicationWillTerminate:` delegate method. In general, you shouldn't use either method to perform operations when the application quits, because, by definition, they delay the quitting operation.

1. Add the `NSSupportsSuddenTermination` key to the `Vermont_Recipes-Info.plist` file. This key isn't yet in the built-in list of keys, so you have to type it in. Set its value to YES. This key does not exist in Leopard, so Leopard simply ignores it.
2. Review the application's operations to see whether it does anything that requires holding off sudden termination, other than the operations that are taken care of by NSUserDefaults, NSDocument, and others.

The diary window controller installs several notification observers. They are removed in `-[DiaryWindowController dealloc]` when the window is closed. If the diary window is open and has no unsaved changes when the user quits or logs out, the application will not have an opportunity to remove them when it is killed. This should not be a problem, however, because the default notification center is a creature of the application. It no longer exists after the application is killed.

A review of the project's code reveals nothing else that could pose a problem if the application suddenly terminates.

3. Save a snapshot. Name it **Recipe 8 Step 9**, and add a comment saying, **Added sudden termination support**.

## Step 10: Internationalize the Application's Display Name

In Steps 9 and 10 of Recipe 1, you configured the application's Info.plist and InfoPlist.strings files with a variety of settings. Two of them are of interest here: `CFBundleDisplayName` and `LSHasLocalizedDisplayName`. In Recipe 1, you set `CFBundleDisplayName` to the name of the application, *Vermont Recipes*, in the `Vermont_Recipes-Info.plist` and

InfoPlist.strings files. However, you left `LSHasLocalizedDisplayName` at its default value of 0, or false, in the `Vermont_Recipes-Info.plist` file. These settings have to do with the “long” application name that the Finder and other applications display when they refer to this application by a human-readable name.

You should allow your application’s display name to be localized if it is descriptive. The proper noun *Vermont* in *Vermont Recipes* might remain unchanged in other languages, but *Recipes* is an ordinary word describing the instructions for preparing a culinary dish or for carrying out any complicated procedure, such as writing an application. Users whose primary language is not English will feel more at home with your application if they see the local word for *Recipes*.

The “Display Names” section of Apple’s *File System Overview* explains that all applications should support display names for a variety of features, including not only the application’s name but also document names, font names, and the names of document types that might appear in file save and open panels. It describes display names as generated names based on the user’s current preferences—specifically, in this case, the Language setting in the Language & Text pane of System Preferences.

In this step, you deal only with the application’s display name, because it is set in the `Vermont_Recipes-Info.plist` and `InfoPlist.strings` files. While the Finder and other applications present the changeable application display name to the user, the underlying file system continues to use the fixed name you provided in the `CFBundleName` setting. The file system routines in Cocoa, for example, use the fixed application bundle name, not the display name. This is why Apple’s documentation warns against using the display name to specify a file’s name, path, or URL in code.

The application’s bundle display name can be retrieved in code and used for display in your application’s dialogs and other places. You retrieve the display name using `-[NSFileManager displayNameAtPath:]`. This method retrieves the localized display name from the application’s `InfoPlist.strings` file if the user has not changed the application’s name in the Finder. You should always retrieve a fresh copy of the display name immediately before displaying it, as the user might have changed the preference setting or renamed the application in the Finder since you last displayed it. The system always gives any user-edited application name precedence over the value set in the `InfoPlist.strings` file.

When you left the value of the `LSHasLocalizedDisplayName` setting to false in Recipe 1, you were not actually turning off the system’s display name feature; you were only making it less efficient. To improve performance when the application uses a localized display name, Apple instructs that you should always set the value of the `LSHasLocalizedDisplayName` key in the `Vermont_Recipes-Info.plist` file to true by selecting the checkbox. Do that now.

# Step 11: Add Application and Document Icons

No application is complete without an application icon and document icons. The system shows these on the desktop, in the Dock, in the Finder's Get Info window, and in various alerts and dialogs. Also, the application normally shows its icon in its About window.

This is not a tutorial on using graphics applications to create images suitable for icons, so you will have to create, find, or buy your own graphics to serve as icons, using whatever resources are available to you. You may find it convenient to use a drawing program, a scanner, or a digital camera to acquire the images, and an image-editing application to edit them. A variety of native Mac OS X applications are available for the purpose. Read the "Icons" section of *Apple Human Interface Guidelines* for guidance on the proper design of icons. The "Creating Icons" subsection has several tips on how to go about creating them. Pay close attention to the first tip: "For great-looking icons, have a professional graphic designer create them."

To obtain suitable graphics for the Vermont Recipes icons, I scanned the cover and a page from an antique Vermont cookbook into Adobe Photoshop. Being a lawyer, I naturally made sure that its copyright had expired. I also scanned a real antique spoon for use as a badge on the application icon. I placed each image on a 1024-by-1024-pixel or larger canvas, resizing the image as needed to leave ample blank space around the image to permit rotation and other effects. I then made the area outside the image transparent. Transparency is important; without it, the icons will have an opaque square background when viewed against a desktop picture. Finally, I saved each image in PSD format so that I could easily return to Photoshop to tinker with them. (In fact, one of the master images is several years old, left over from the first edition of this book.) Icons are supposed to be somewhat abstract, and, strictly speaking, these images may be too photorealistic. I'm not an artist, so they will have to do. The master PSD images are available for download from the book's Web site, in case you want to follow along in this step.

The largest icon currently supported by Mac OS X is 512 by 512 pixels. A document icon is supposed to have the top-right corner turned down, with a smaller image in the center identifying it graphically as being related to the application. For the best results, start with an existing blank document icon sized at 512 by 512 pixels. A brief search turned up the GenericDocumentIcon icon in Apple's CoreTypes.bundle, which is located on your computer at /System/Library/CoreServices/CoreTypes.bundle. Using the Finder's contextual menu on CoreTypes.bundle, I chose Show Package Contents. Then I opened the Contents and Resources folders and located the GenericDocumentIcon.icns file. I opened it in Preview. In Preview's sidebar, I selected the

largest image and, from the contextual menu, chose Save a Copy to Folder and saved it. I ended up with a file named `GenericDocumentIcon.tiff` with a resolution of 512 by 512 pixels. I opened it in Photoshop and removed the black mask outline, and it was ready for use as the background of the document icons.

The main document, which opens automatically when you launch Vermont Recipes, is the recipes document. In accordance with the *HIG*, I composed its icon by placing a smaller image of the application icon—the cover of the Out of Vermont Kitchens cookbook—in the center of the blank document image, after removing its badge, the image of an antique spoon. Vermont Recipes also owns another document type, the Chef’s Diary document. For its icon, I placed an image of another page from the Out of Vermont Kitchens cookbook on a blank document image. This page appears similar to the cover used on the application and main document icons, so it maintains a consistent graphical theme.

Once I finished creating the master application and document icon images as PSD files, I produced smaller Photoshop images with sizes of 16, 32, 48, 64, 128, 256, and 512 pixels square. Icons, unlike images used elsewhere in an application, are bitmap rather than vector graphics, so you supply separate images in several sizes to be substituted as the user scales the icon between larger and smaller sizes. Ideally, you optimize each of them, particularly the smallest images, to look good at its size. I saved them in PNG-24 format with transparency. These images are also available for download from the book’s Web site.

Only some of these images are used in the application and document icons. The 48- and 64-pixel images are for use in the Vermont Recipes help book you will create later. They may also come in handy when you build a Web site to promote your application.

Once you have the PNG image files for each icon size in hand, you are ready to turn them into icons. In this recipe, you use the Icon Composer application provided with Apple’s Developer Tools to combine your images into three icon files, one for the application, one for the main recipes document, and one for the Chef’s Diary document.

Launch Icon Composer. It’s in the `/Developer/Applications/Utilities` folder.

1. Using an untitled Icon Composer window, drag application PNG images onto the empty squares, matching the indicated size. Icon Composer wants images at 512, 256, 128, 32, and 16 pixels.

When you’re done, you can examine the mask that Icon Composer created automatically by choosing Masks in the segmented control at the bottom of the Icon Composer window. The masks define the area where a user’s click is effective. These generated masks appear to be fine.

You can also use Icon Composer to examine a preview of the icon’s appearance in the real world. Click Preview in the segmented control, and experiment

with the slider to see how smoothly the icon resizes. Also use the pop-up menu to examine and resize it against different backgrounds. It is especially important to make sure the application icon looks right against a black background such as that used in the Finder's Cover Flow view. Icons without light borders can lose some of their elements. Your application icon works, because the spoon badge has a reflective edge around it, although a graphic artist could make some improvements.

2. Choose File > Save As, give the icon file a name, designate its location, and click Save. The file is automatically given the required .icns extension.
3. You can double-click the saved icon file to open it in Preview and examine it.
4. Drag copies of the saved icon files into the root level of the project folder. For Vermont Recipes, name the application icon **VRApplicationIcon.icns** and the document icons **VRDiaryDocument.icns** and **VRRecipesDocument.icns**. These go in the root project folder rather than the English.lproj folder because icons are not localized.
5. Now make suitable entries in the Vermont\_Recipes-Info.plist file. When you initially set up the Info.plist file in Recipes 1 and 3, you left the icon entries blank until later, so you have to provide them now.

Open the Vermont\_Recipes-Info.plist file, and in the CFBundleIconFile entry that is already there, enter **VRApplicationIcon**. You have the option to enter VRApplicationIcon.icns, with the file extension, but you don't have to.

6. Verify that it has become the application icon by opening the Vermont Recipes target Info inspector and selecting the Properties tab. You see that the Icon File entry now indicates that VRApplicationIcon is the icon file, and you see its image in the image well to the right. You could have entered it here in the first place, instead of entering it in the Vermont\_Recipes-Info.plist file (**Figure 8.15**).



**FIGURE 8.15** The Vermont Recipes application icon.

7. While you're still in the Target Properties tab, go to the DocumentTypes section at the bottom of the Target Info window and scroll right to find the Icon File column. In the two entries for the RecipesDocument type, enter **VRRecipesIcon**. In the two entries for the DiaryDocument type, enter **VRDiaryIcon**. Again, you have the option to include the file extension, but you don't have to.

8. Verify that they have become the document icons by opening the `Vermont_Recipes-Info.plist` file again. You see that each of the items in the `CFBundleDocumentTypes` array for the `CFBundleTypeIconFile` key now shows the name of the appropriate icon file in the Value column. You could have entered these keys and values in the `Vermont_Recipes-Info.plist` file in the first place (**Figure 8.16**, **Figure 8.17**).



**FIGURE 8.16** The Vermont Recipes recipes document icon.



**FIGURE 8.17** The Vermont Recipes Chef's Diary document icon.

9. You also need to add one icon file reference to the `UTExportedTypeDeclarations` array in the `Vermont_Recipes-Info.plist` file. Add an entry for the `UTTypeIconFile` key, setting its value to **VRDiaryIcon**. Again, the file extension is optional. For every custom document type that you create and that your application owns, you should export all of the `UTExportedTypeDeclarations` keys. Don't add entries for the other document types, as Vermont Recipes doesn't own them.
10. Build and run the application. You see the new application icon in all the expected places. For example, it appears in the Dock, and when you open the About window, you see it there too. When you save a Chef's Diary document, its new document icon appears as well.

If you don't see the diary document's icon but instead see a white document image with the file extension `VRDIARY` emblazoned on it, you probably saved the document in a folder where you had previously turned on icon previews. The icon preview trumps the document's icon. Turn off icon preview mode to see the document's icon by deselecting the "View icon preview" checkbox in the dialog opened by choosing `View > View Options` in the Finder.

11. Gather all of the Photoshop, PNG, and icon files you created in this step, and save them in a folder alongside the Vermont Recipes project folder. If you revise the Vermont Recipes application in the future, you may want to make changes to the icons. It would be a shame to have to re-create them from scratch.



# Step 12: Enable the Application to Run Under Leopard

You have been developing Vermont Recipes under Mac OS X 10.6 Snow Leopard, and since Snow Leopard runs only on Intel-based Macintosh computers, that's the hardware you have been using. The application specification requires that it be able to run under Mac OS X 10.5 Leopard as well. Because Leopard can run on PowerPC-based as well as Intel computers, it should also be able to run on PowerPC Macs.

Unless you have changed some of the project's settings, the application is currently incapable of running on PowerPC Macs. In Recipe 1, you configured the project's Mac OS X deployment target build setting to allow it to run under Mac OS X 10.5 Leopard, but that isn't enough. You don't have to try it to see that this is so. Just open the project's build folder and open a Get Info window on the application's icon. It says that the file's kind is Application (Intel). By default, a new project runs only on Intel hardware.

Even when you change the appropriate settings to enable the application to run on PowerPC Macs, it has some problems under Leopard that you will fix in this step.

1. Start by enabling the application to run on PowerPC hardware.

Open the project's Info window and choose the Build tab to review the settings you have been using for development. In the Debug configuration, look at the Architectures section. It specifies a Base SDK (or SDKROOT if you use the contextual menu to set it to display setting names instead of setting titles) of Mac OS X 10.6 (or `macosx10.6` if you set it to display definitions instead of values). As explained in Recipe 1, this is necessary so that you can build the latest Snow Leopard features into the application during development. Further down, in the Deployment section, you already set the Mac OS X Deployment Target (or `MACOSX_DEPLOYMENT_TARGET`) to Mac OS X 10.5 (or 10.5). Back up in the Architectures section, you turned on Build Active Architecture Only (or `ONLY_ACTIVE_ARCH`) by selecting the checkbox (or setting the value to YES). This means that the executable code generated when you built the application in the Debug configuration is designed to run on either the i386 architecture or the x86\_64 architecture, depending on your development machine. Those are the two Valid Architectures (`VALID_ARCHS`) set by default. The Architectures (or `ARCHS`) setting is "Standard (32/64-bit Universal)" (or `$(ARCHS_STANDARD_32_64_BIT)`), but that doesn't affect the other settings for purposes of development because the Build Active Architecture Only (or `ONLY_ACTIVE_ARCH`) setting overrides it.

Now change the Configuration setting to Release. Everything is the same, except that the Build Active Architecture Only (or `ONLY_ACTIVE_ARCH`) setting has been turned off (set to NO). This means that your release builds will be capable of running in 32-bit or 64-bit mode (if the architecture supports it) on either

kind of Intel Mac. But your release build will not be able to run on PowerPC Macs because Valid Architectures (or `VALID_ARCHS`) is still limited to i386 and x86-64 hardware.

If you have a PowerPC Mac handy, try it. Build the application for release based on the Vermont Recipes target, and copy the built application from the project's build folder on your development machine to the PowerPC Mac's Applications folder. The first thing you notice is that the application's icon has a circle and slash emblem superimposed on it, indicating that it can't run here. Use the contextual menu to open its Get Info window for a better look at the icon preview. You see that the kind is still listed as Application (Intel) (**Figure 8.18**). If you double-click the application icon to run it, an alert confirms that this is forbidden (**Figure 8.19**).



**FIGURE 8.18** The unmodified application's Info window under Leopard on a PowerPC Mac.



**FIGURE 8.19** An alert warning that the unmodified application cannot be run on a PowerPC Mac.

To enable the application to run on PowerPC equipment, all you have to do is add the appropriate architecture to the Valid Architectures (or `VALID_ARCHS`) setting. The available choices are documented in the *Xcode Build Setting Reference*. Search for `VALID_ARCHS` to find them. The one you want is `ppc`.

Remember that the new Vermont Recipes SL target runs only on Snow Leopard, and PowerPC hardware cannot run Snow Leopard applications. It therefore would make no sense to add the `ppc` architecture in the project Info window, since both the Vermont Recipes and Vermont Recipes SL targets would inherit it. It also wouldn't make sense to add it in the Vermont Recipes SL target info window. Instead, add it only in the Vermont Recipes target's Info window.

Open the Vermont Recipes target's Info window now, choose the Release configuration, and in the Build tab, double-click the Valid Architectures (or VALID\_ARCHS) setting. A sheet opens listing the existing values (**Figure 8.20**). Click the Add (+) button, enter **ppc**, and click OK. The value of the setting is now the string "ppc i386 x86\_64" with the individual architectures separated by spaces, and the release configuration of the Vermont Recipes target will run under Leopard on PowerPC as well as Intel equipment. There is no need to add ppc to the Debug configuration, because you develop the application only under Snow Leopard on Intel equipment.

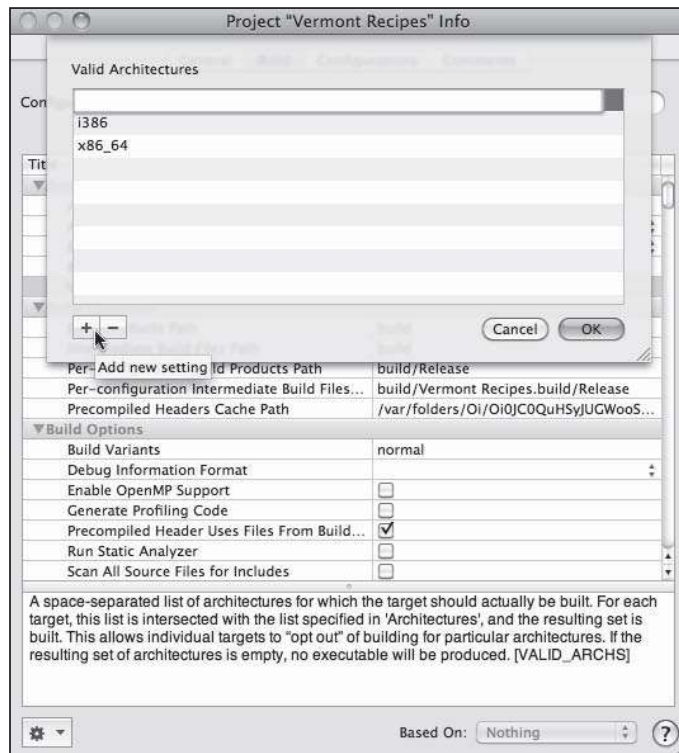


**FIGURE 8.20**  
The project build settings before adding the ppc architecture.

If, like me, you own a 64-bit-capable Power Mac, you might wonder why I haven't suggested adding ppc64 to the Valid Architectures. Sadly, Xcode in Snow Leopard doesn't allow you to build for the 64-bit PowerPC architecture. It probably doesn't matter, because all indications are that the 64-bit PowerPC architecture wouldn't generally provide a reliable speed increase sufficient to justify the effort.

Before trying to run the application under Leopard on a PowerPC Mac again, you should fix a couple of problems with the nib files.

2. In Recipe 2, when you first started setting up the application's GUI using Interface builder, you did not pay much attention to the environment in which Vermont Recipes is expected to run. Depending on your nib file settings, you might have noticed a while ago that a warning was being generated from time to time when you built the project. Usually, the Xcode Build Results window reported "No issues" after every build. But on those occasions when you cleaned the project or revised some nib file settings before building it, the Build Results

window might have reported one warning on the subsequent build, to the effect that “The ‘Pane Splitter’ divider style is not supported on Mac OS X versions prior to 10.6.” This warning would have appeared only if you had set one of the nib files to run under Leopard as well as Snow Leopard.

In fact, you should now set all of the nib files to run under Leopard as well as Snow Leopard. You should also fix the Pane Splitter problem and one other problem of the same nature.

Open the MainMenu nib file; then in Interface Builder choose Window > Document Info. You see that, by default, the nib files in the template from which you created the project in Recipe 1 set the deployment target of the new nib files to Mac OS X 10.6 and the development target to Interface Builder 3.0.

You plan to release Vermont Recipes for Mac OS X 10.5 Leopard, as well as Snow Leopard. The “Testing and Validation” section of the *Interface Builder User Guide* states that the deployment target for a nib file should match the deployment target for the Xcode project. The Xcode project’s deployment target for Vermont Recipes is Mac OS X 10.5. Apparently, Interface Builder doesn’t match its deployment target with the Xcode deployment target automatically, although the documentation seems to suggest that it might under some circumstances. Change the MainMenu nib file’s deployment target to Mac OS X 10.5 now.

The Development Target of the Main Menu nib file should be listed as Interface Builder 3.2. Interface Builder is up to version 3.2.1 as I write this, and from the beginning I have recommended doing all of your Vermont Recipes development work on Snow Leopard. Change the MainMenu nib file’s development target to Default - Interface Builder 3.2, if it isn’t already set that way.

No sign of an error appears in the table at the bottom of the window, so the Main Menu nib file is apparently OK. Save and close it.

Next, open the RecipesWindow nib file and choose Window > Document Info. The deployment target here should also be set to Mac OS X 10.5 and the development target set to Default - Interface Builder 3.2. When you set the deployment target, the table at the bottom of the window reports the same warning as the warning I saw during some of my builds: “The ‘Pane Splitter’ divider style is not supported on Mac OS X versions prior to 10.6.” It looks as though the “Pane Splitter” divider style won’t work right when Vermont Recipes is run under Leopard.

Next, open the Diary Window nib file and choose Window > Document Info. Again set the deployment target to Mac OS X 10.5 and the development target to Default - Interface Builder 3.2. You see two problems listed in the table at the bottom. The same “Pane Splitter” problem appears in this file as in the RecipesWindow nib file. In addition, the search field apparently uses a single line mode that is not supported under Leopard.

If the images you created in Recipe 4 for the navigation buttons were not 24 by 24 pixels or smaller when you created them, you would also see four “Clipped Content” problem reports relating to the navigation buttons in Interface Builder’s Document Info window. You don’t see them because you sized both the images and the button views to 24 by 24 pixels. If you were to run the application under Leopard with images that were larger than the views, the images would become distorted as you resized the diary window (**Figure 8.21**). The only cure would be to redraw the images at 24 by 24 pixels. Even if you used 24-by-24-pixel images, you may see four clipping warnings in Xcode when you build the project. This is reportedly due to a bug in Xcode.



**FIGURE 8.21** Distorted navigation buttons when running under Leopard with oversized images.

3. To fix the two nib file problems you found, it’s best to add some code. My approach is to set the nib file to use only those features that are available in the oldest version of Mac OS X that my application supports, the deployment target, and then turn on the newer features in code if the application detects that it is running under the newer version of the operating system. This way, the capabilities of my application are optimized for users whose operating systems are up to date, but it works appropriately when running on older operating systems.

Start with the search field issue, since it affects only one document, the Chef’s Diary. A quick way to find the Cocoa method at issue is to open the DiaryWindow nib file, select the search field in the window’s design surface, and open the Validated Diary Search Field Attributes inspector. There, you see a checkbox labeled Uses Single Line Mode. Pause the pointer over that checkbox, and in a moment a help tag appears, identifying the relevant method as `-usesSingleLineMode:`. Search for that method in the Xcode documentation window, and you find its entry in the *NSCell Class Reference*. Sure enough, it is marked as having been introduced in Snow Leopard.

To fix this problem in the nib file, deselect the Uses Single Line Mode checkbox and save the nib file. Then return to Xcode, and in the DiaryWindowController.m implementation file, add this code at the beginning of the `-windowDidLoad` method:

```
if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 [[[self searchField] cell] setUsesSingleLineMode:YES];
}
```

An alternative way to test whether it is safe to execute this code is to check whether the method is available in the Cocoa frameworks at run time, like this:

```
if (self respondsToSelector:@selector(setUsesSingleLineMode:)) {
```



However, in a moment you will place in the same block some additional code that does not rely on the `-setUsesSingleLineMode:` method but must still be confined to Snow Leopard. So don't use the `-respondToSelector:` technique here.

4. Still in the `DiaryWindow` nib file, deal with the pane splitter problem. In the diary window design surface, select the split view divider, and then open the Split View Attributes inspector. You see that the Style pop-up menu is set to Pane Splitter, which the error messages indicate is not available in Leopard. When you pause the pointer over the pop-up menu, the help tag indicates that the relevant Cocoa method is `-dividerStyle`.

In the *NSSplitView Class Reference*, there is a section describing the “Split View Divider Styles,” indicating that three possible divider styles are available in Snow Leopard. One of them, `NSSplitViewDividerStylePaneSplitter`, is marked as having been introduced in Snow Leopard. Anytime you encounter a new feature introduced in Snow Leopard, it is worth looking at the Snow Leopard release notes. In this case, you find in the *Mac OS X SnowLeopard Release Notes: Cocoa Application Framework* a section headed “New NSSplitView pane splitter divider style.” It indicates that the *Apple Human Interface Guidelines* “strongly encourage you to migrate away from the ‘thick’ style to this new style,” and indeed the *HIG* does strongly encourage you to do so.

Change the pop-up menu setting to “Thick divider” and save the nib file. Before you close the file, choose Window > Document Info again to see what has changed. The “Pane Splitter” issue is now gone from the list, but the four Clipped Content problems remain. Just for the heck of it, temporarily change the deployment target setting back to Mac OS X 10.6. A new warning appears, saying that the split view has a “Discouraged Configuration,” with the further explanation that “The ‘Thick Divider Style’ is discouraged on Mac OS X versions after 10.5.” You already knew that, and you will now fix the problem in code. Make sure you set the deployment target back to Mac OS X 10.5 and save the nib file before closing it.

In the `DiaryWindowController.m` implementation file, go to the `if` block you just added at the beginning of the `-windowDidLoad` method, and add this statement:

```
[[self splitView] setDividerStyle:NSSplitViewDividerStylePaneSplitter];
```

The `splitView` instance variable already exists, as does its getter method.

5. Go to the `RecipesWindowController` nib file and the `RecipesWindowController.m` implementation file, and make similar changes to fix the Pane Splitter divider issue there. You should leave the thin divider in the recipes window alone.



This implementation file does not yet have accessor methods for the vertical split view, so create them now, copying the instance variable and accessor methods for the diary window controller's split view but naming this one `verticalSplitView`. Don't forget to connect the IBOutlet in Interface Builder. The statements to add to the `-windowDidLoad` method are these:

```
if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
 [[self verticalSplitView]
 setDividerStyle:NSSplitViewDividerStylePaneSplitter];
}
```

The application is now in compliance with the *HIG* under Snow Leopard, and known problems under Leopard have been resolved.

6. Build the application for release now in order to test it under Leopard on a PowerPC computer. Be sure to use the Overview pop-up menu to make the Vermont Recipes target the active target and to make Release the active configuration. If you don't have a PowerPC Mac running Leopard, you'll have to take this on faith, but for a real-world application, you should always find some way to test it on all hardware you claim it supports.

This time, when you move the built application to a PowerPC Mac, its icon appears without the circle-and-slash badge. In its Info window, its kind is reported as Application (Universal) (Figure 8.22).



**FIGURE 8.22** The Vermont Recipes Info window after enabling Leopard support.

Launch the application under Leopard on the PowerPC computer. It starts right up and opens the recipes window by default. Everything seems to work as expected. Create a new Chef's Diary window. The Add Tag button in the diary window is not dynamic and does not change to Tag All when you hold down the Option key, but the Add Tag menu item in the Diary menu remains dynamic.

## Step 13: Build and Run the Application

You have two applications to build and run this time. One should be built from the Vermont Recipes SL target and tested on an Intel-based Mac running Snow Leopard. The other should be built from the Vermont Recipes target and tested on three different platforms: a PowerPC Mac running Leopard, an Intel Mac running Leopard, and an Intel Mac running Snow Leopard. The version of the application built for Leopard presumably will not be recommended for Snow Leopard, because the version of Vermont Recipes meant to be run on Snow Leopard has an additional feature and more efficient code. Nevertheless, a user running Leopard may well upgrade to Snow Leopard without immediately installing the Snow Leopard version of Vermont Recipes, so it has to run correctly on Snow Leopard as well as Leopard.

For a valid testing experience, remove any leftover autosaved files from ~/Library/Autosave Information and other locations where you saved Vermont Recipes documents, and remove the preferences file from the Preferences folder.

While testing each version of Vermont Recipes on all appropriate platforms, run through all of the new features you added in this recipe, including the Save As PDF menu item, the alternating Show Recipe Info and Hide Recipe Info menu items, the dynamic Add Tag and Tag All menu items and button, autosaving and restoring documents after a crash, help tags, VoiceOver support, and the default name for saving new diary documents.

## Step 14: Save and Archive the Project

Quit the running application. Close the Xcode project window, discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like **Vermont Recipes 2.0.0 - Recipe 8.zip**. The working Vermont Recipes project folder remains in place, ready for Recipe 9.

# Conclusion

The Chef's Diary and its supporting GUI and menu structure are now finished, and apart from the recipes document, the application is in very good shape. You may not have noticed all of the common Macintosh application features that it already supports. For example, because the Chef's Diary is based on the standard RTF format, Quick Look and Spotlight already work. Try them out.

To prove it to yourself, launch Vermont Recipes, create a new diary document, and add several diary entries and a bunch of tags. Also type some text that includes unique words to search for, such as **Rumpelstiltskin**, **Lubber Fiend**, and **antidisestablishmentarianism**. Then save the file and close it, and quit Vermont Recipes for good measure.

Select the saved file in the Finder and press the Spacebar. A Quick Look preview opens, showing an image of the contents of the file.

Deselect the file, and then press Command-Spacebar. The Spotlight search field opens. Enter **Lubber Fiend**. On my computer, the Chef's Diary.vrdiary file appears as the second entry in the list of found documents, after the autosave recovery version of the file. (Surprisingly, several other files also appear in the list, all from OmniObjectMeter example files.)

There isn't much left to do before you can consider Vermont Recipes a finished product—apart from the recipes document. In upcoming recipes, you will add a few things that almost every application should have, such as printing support, a preferences window, a help book, and AppleScript support. Then, at the end of Section 2, you will prepare the application for deployment.

## DOCUMENTATION

---

Read the following documentation regarding topics covered in Recipe 8.

### **Class Reference and Protocol Documents**

*NSDocument Class Reference*

*NSPrintOperation Class Reference*

*NSMenuItem Class Reference*

*NSMenu Class Reference*

*NSMenuDelegate Class Reference*

*NSEvent Class Reference*

*(continues on next page)*

## DOCUMENTATION *(continued)*

---

### **Class Reference and Protocol Documents** *(continued)*

*NSNotificationCenter Class Reference*

*NSAccessibility Protocol Reference*

*NSProcessInfo Class Reference (Sudden Termination)*

*NSSavePanel Class Reference*

### **General Documentation**

*Mac OS X SnowLeopard Release Notes Cocoa Application Framework*

*Blocks Programming Topics*

*Introducing Blocks and Grand Central Dispatch (Block Objects)*

*Mac OS X SnowLeopard Release Notes Cocoa Foundation Release Framework*

*Technical Note TN2064: Ensuring Backwards Binary Compatibility - Weak Linking and Availability Macros on Mac OS X*

*Online Help (Tooltips)*

*Interface Builder User Guide (Configuring User Assistance Attributes, and Using Mac OS X Technologies)*

*Accessibility Programming Guidelines for Cocoa*

*Accessibility Overview*

*File System Overview (Display Names)*

*Cocoa Application Tutorial*

*Information Property List Key Reference*

*Uniform Type Identifiers Overview*

*Simplifying Data Handling with Uniform Type Identifiers*

*Xcode Build Setting Reference*

### **Third-Party Commentary**

*Basic Blocks—[www.Friday.com/bbum/2009-08-29/basic-blocks/](http://www.Friday.com/bbum/2009-08-29/basic-blocks/)*

*^ Blocks Tips & Tricks—[www.Friday.com/bbum/2009/08/29/blocks-tips-tricks/](http://www.Friday.com/bbum/2009/08/29/blocks-tips-tricks/)*

*Beginners Guide to Blocks in Cocoa—[www.degutis.org/dev/2009/08/30/beginners-guide-to-blocks-in-cocoa/](http://www.degutis.org/dev/2009/08/30/beginners-guide-to-blocks-in-cocoa/)*

*Practical Blocks—[www.mikeash.com/?page=pyblog/Friday-qa-2009-08-14-practical-blocks.html](http://www.mikeash.com/?page=pyblog/Friday-qa-2009-08-14-practical-blocks.html)*

*Programming with C Blocks—<http://thirdcog.eu/pwcblocks/>*

# Add Printing Support

Unlike Recipe 8, which covered many disparate topics, this recipe has a single focus: printing.

Cocoa printing underwent profound changes in Mac OS X 10.5 Leopard, both in the API and in the user interface. The primary Cocoa printing documentation, *Printing Programming Topics for Cocoa*, has not yet been updated to reflect these changes, so for details you must also read the extensive discussion of the printing API in the Leopard release notes, *Mac OS X Developer Release Notes: Cocoa Application Framework (10.5 and Earlier)*.

One of the more significant changes from the user's perspective is that the familiar page setup panel is no longer needed for many applications. The Print panel now provides convenient one-stop shopping, giving you complete control over all aspects of the printing experience in a single panel, including all of the features that users formerly set in the page setup panel. In addition, the Print panel in Leopard and Snow Leopard now provides full support for print previews.

A Page Setup panel may still be appropriate in an application that allows the user to save document-specific print settings with each document, as TextEdit does, for example. Even in that case, however, the settings in the Page Setup panel can be included in the Print panel, and with a little code you can save them on a per-document basis. A Page Setup panel is not needed for the Chef's Diary, because it is a one-of-a-kind document. You save appropriate print settings for it in the application's user defaults so that they remain in effect every time the user prints the document, unless the user changes them.

The organizing principle underlying the Print panel is to make the most frequently changed settings available in one location, in the upper-right corner of the panel.

## HIGHLIGHTS

*Adding support for printing to a document-based application*

*Modifying the standard settings displayed in the Print panel*

*Creating a custom print accessory view*

*Creating an accessory view controller*

*Creating a singleton object using a factory method*

*Using NSObject's +initialize method*

*Setting initial print settings in the user defaults database*

*Setting changed print settings in the user defaults database*

*Creating a print view*

*Printing headers and footers*

*Scaling the printout*

There, you typically choose such things as the printer to use, a named set of presets defaulting to Standard, the number of copies to print, whether to print all or a range of pages, the paper size, its orientation, and its scaling factor. Some of these, such as paper size and orientation, formerly appeared only in the Page Setup panel. Below this is the accessory view area, which displays several groups of settings provided by the system and even by individual printers. The settings shown in the accessory view area are controlled by the features pop-up menu. This menu can, optionally, include one or more custom accessory views for the specific application. When a single custom accessory view is provided, the system displays it by default when the Print panel is opened, and it usually is named for the application. To the left is a preview of the printed document. The preview allows you to flip through all of the pages to be printed, and its appearance changes interactively to display every change you make to the settings. There is also an area at the bottom of the Print panel containing additional features, including many options for saving *digital paper* in the PDF pop-up menu. For an example of a typical Print panel, look at the TextEdit Print panel (**Figure 9.1**).

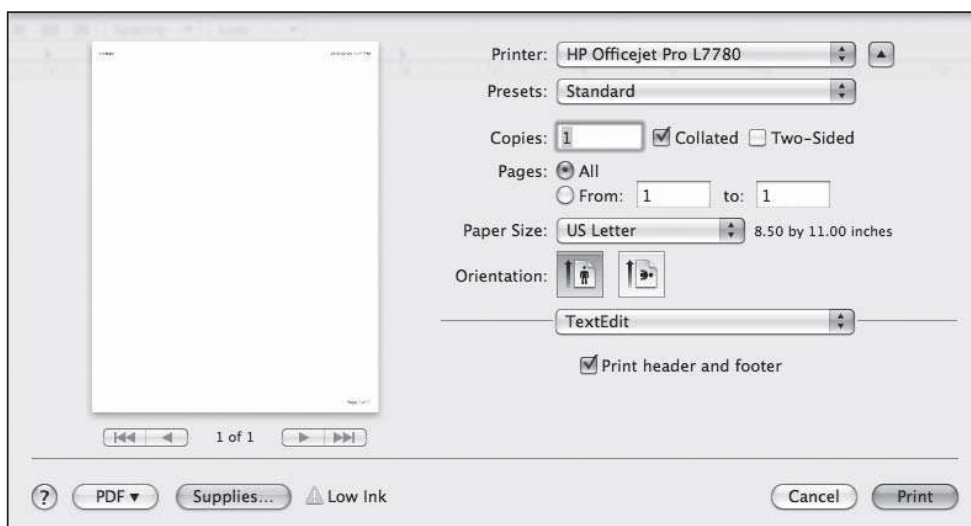


**FIGURE 9.1**  
The TextEdit  
Print panel  
under Snow  
Leopard.

A key element in the Print panel is the features pop-up menu that appears in the center of the settings area. In most applications, including TextEdit, the default menu item shows the name of the application, and several application-specific settings appear below it in an area called the *accessory view*. The pop-up menu includes a number of other menu items that display built-in views for various groups of settings, such as layout and paper handling. The printer supplies some of these, and they therefore change when you choose a different printer.

The last time you tried it, the Print menu item in the File menu did not work. The Print menu item was properly connected to NSDocument’s built-in `-printDocument:` action method in the MainMenu nib file, thanks to the document-based application template from which you generated the Vermont Recipes project. However, when you chose File > Print while the Chef’s Diary document was active, the Debugger Console displayed an error and the document did not print.



Try to print the diary document again now. Create a new diary document, type something in it, and choose File > Print. Miraculously, the document prints! The explanation is simple: The error message generated earlier reported that “printOperationWithSettings>Error: is a subclass responsibility but has not been overridden.” In Recipe 8, you did override it in the DiaryDocument class while setting up the new Save As PDF menu item. As an unexpected side effect, you can now print the diary document. You’re treated to a fairly complete Print panel (Figure 9.2).

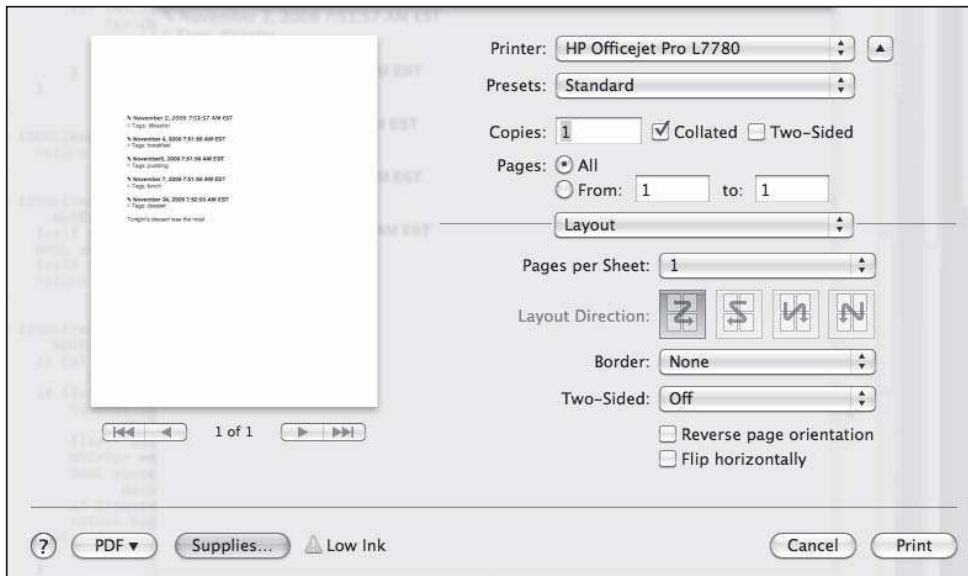


**FIGURE 9.2**  
The Chef's  
Diary Print  
panel before  
adding custom  
features.

If you have read the “Printing Documents” section of *Printing Programming Topics for Cocoa*, you might wonder how this could work. That document specifically tells you that you must override NSDocument’s `-printShowingPrintPanel:` method to create and run a print operation. Read the more up-to-date *NSDocument Class Reference*, however, and you learn that this method was deprecated way back in Mac OS X 10.4 Tiger. According to the class reference, the `-printDocument:` action method now automatically calls `-printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:`, which in turn automatically calls the `-printOperationWithSettings:error:` method that you overrode in Recipe 8. According to the class reference, the `-printOperationWithSettings:error:` method does nothing by default, and you must override it to print the document. You did this in Recipe 8.

That isn’t the end of this recipe, however. You still have a lot of work to do.

For one thing, the features pop-up menu in the middle of the Print panel’s settings area defaults to the Layout view, not to a Vermont Recipes view. The print panels in many applications similarly omit a custom accessory view, and this is fine if you don’t need to give the user the ability to configure unique application features for printing. Here, however, there are a few customizations a user might like when printing the Chef’s Diary. For example, it may eventually contain many entries, and the user might want to print only some of them. You will therefore add a custom

accessory view to the Print panel. It will allow the user to print the entire document, or only the current entry, or only the currently selected text. It will also allow the user to exclude tag lists when printing, and to include headers and footers, as well as some other custom features.

The printed Chef's Diary currently looks just like the document you see on the screen. It contains the text you typed into the diary, and the entry titles and tag lists, but it does not contain a header or a footer, and it prints no page numbers. You add these features and others in this recipe. The PDF document that you save with the Save As PDF command does not reflect these settings, but you can save a PDF version of the document that does include them by choosing Save as PDF from the PDF pop-up menu in the Print panel.

The key to gaining control over the print settings in Cocoa is the print info object. When the user changes the settings in the Print panel, the new settings are reflected in the print info object. When the user clicks Print, a subclass of `NSView` that you write in this recipe uses the information in the print info object to format the contents of the view for printing.

In this recipe, you implement full custom printing support for the Chef's Diary document.

## Step 1: Create a Print Panel Accessory View in Interface Builder

Start by creating a custom accessory view for the Chef's Diary. It will be visible to the user in the Print panel when the user chooses Vermont Recipes Chef's Diary in the features pop-up menu. The user interface elements in the accessory view are tailored to the Chef's Diary.

First, you allow the user to print the entire document, only the current entry, or the text currently selected in the document. This way of dividing up the parts of the document to print is inconsistent with the notion of printing a range of pages, so you remove the page range settings from the main print settings area in the Print panel. Although the user can see the page numbers in the preview, Vermont Recipes does not display page numbers in the diary window, unlike a typical word processing application. Page numbering in the printed document varies depending on the size of the paper, which parts of the document the user chooses to print, and whether to include tag lists in the printout.

In addition, you allow the user to omit tag lists from the printout and to include headers and footers. Tag lists are turned off by default because they are intended primarily for

live searching. Headers and footers are turned on by default. The footer includes a timestamp, and you enable the user to choose whether the date is the date of printing or the date the document was last saved. The date it was printed is the default.

1. Leave the archived Recipe 8 project folder where it is, and open the working Vermont Recipes subfolder.

Increment the Version in the Properties pane of the Vermont Recipes target's information window from 8 to **9** so that the application's version is displayed in the About window as *2.0.0 (9)*.

2. The easiest way to create the user interface for a simple accessory view is to use Interface Builder.

In Xcode, choose File > New File. In the New File dialog, select User Interface in the source list on the left. In the upper-right pane, select View XIB (**Figure 9.3**).



**FIGURE 9.3** Creating a new View XIB file in Xcode.

Click Next and name the file **DiaryPrintPanelAccessoryView.xib**. Be sure to select both the Vermont Recipes and Vermont Recipes SL target checkboxes, because it needs to be copied into both targets when you build the application. If you forget to do this, you can always come back later and drag the nib file into the Copy Bundle Resource build phase of the Vermont Recipes SL target.

When you click Finish, the nib file is created. Make sure the file is located in the English.lproj folder. If necessary, drag it into the Resources group in the Groups & Files pane, and place it with the other nib files.

Now double-click the file in the Groups & Files pane, and it opens in Interface Builder. You see the familiar nib file document window with File's Owner, First Responder, and Application proxy icons, as well as a Custom View icon. The custom view is open in its design surface, ready for you to add user interface elements.

Before you do that, choose Window > Document Info, and then set the Deployment Target to Mac OS X 10.5 and the Development Target to Default – Interface Builder 3.2.

3. There should be two distinct sections to the Vermont Recipes custom accessory view, one at the top that applies only to the current print job, and one at the bottom that applies to all print jobs. The settings in the top section are reset to their default values every time the user opens the Print panel, while the settings in the bottom section are saved in the user defaults and restored every time the user opens the Print panel.

Drag a label from the Inputs & Values section of the Library window, and drop it in the upper-left corner of the accessory view's design surface. Rename it **This Print Job**. Then drag a horizontal line from the Layout Views section of the Library window, and drop it to the right of the label. Use the arrow keys to position it. I leave a gap of 2 pixels between the label and the horizontal line, and move the horizontal line vertically until it aligns with the bottom of the text in the label. Don't worry about the length of the horizontal line for now.

4. Add another label and a radio group below the section label to allow the user to print the entire document, the current entry, or selected text. Rename the new label **Print:** (with the trailing colon). After dropping the radio group adjacent to the label, select it, and in the Matrix Attributes inspector, change the Rows field from 2 to 3. The new radio button appears at the top, so you'll have to drag the whole group down to align the new top radio button with the label. Change the names of the radio buttons, from top to bottom, to **All entries**, **Current entry**, and **Selection**. With the radio group selected, choose Layout > Size To Fit to make all of the new titles visible.

Arrange to mark the "All entries" button as the selected button. To do this, click the "All entries" button twice to select its cell; then select the State checkbox in the Visual section of the Button Cell Attributes inspector. If one of the other radio buttons also appears to be selected—that is, it has a dot inside the circle to mark it as selected—select its cell in the same way and deselect its State checkbox. The radio group is now set to print all entries by default.

5. Add a label and a horizontal line below the radio group, and name the label **All Print Jobs**.
6. Add a label and a checkbox below the All Print Jobs section header to allow the user to choose whether to print tag lists. Rename the label **Print:** (with the trailing colon). After dragging the checkbox object from the Inputs & Values section of the Library window to the design surface, change its title to **Tag lists**. In the Button Attributes inspector, deselect the State checkbox. The "Tag lists" checkbox is now deselected, and by default tag lists will not be printed.

7. Add another checkbox below the first checkbox to allow the user to choose whether to print headers and footers. You don't need another label, because it is clear that the Print label beside the "Tag lists" checkbox applies here as well. Name this checkbox **Headers and footers**. Leave it selected so that, by default, headers and footers will be printed.
8. Add another label and radio group at the bottom to allow the user to choose whether the timestamp in the header represents the time when the Chef's Diary was printed or when it was last saved. The label should be **Timestamp:** (with the trailing colon), and the two radio buttons should be titled **Date printed** and **Date saved**. Change the selected radio button to the "Date printed" button by changing the State settings as you did with the first radio group.
9. Arrange the user interface elements to comply with the *Apple Human Interface Guidelines (HIG)*, using the Interface Builder guides to help. The two section labels should be at the left margin of the view, while the settings within the two sections should be centered horizontally in the view.

Once the two section headers are placed at the left margin, resize the view horizontally to leave a pleasing amount of white space on either side of the settings' UI elements. Then extend the length of the two horizontal lines adjacent to the section labels to the right margin of the view.

Drag all of the settings elements left or right as needed in order to leave the checkboxes and the radio buttons aligned down the middle, so that the user can move the pointer straight down the center line to select or deselect settings. To get the alignment right, it is helpful to add a custom guide to the design surface. Choose Layout > Add Vertical Guide, drag the new vertical line against the left edge of whichever radio button or checkbox is most perfectly centered, and then line up the other radio buttons and checkboxes so that they are positioned against the guide. If Snap To Guides is checked in the Layout menu, you will probably have to use the arrow keys on the keyboard to get the alignment right. Then remove the vertical guide by dragging it off the design surface.

You may have to adjust the centering of the settings elements. Select all of the settings elements, leaving the headings elements deselected. Move the pointer into white space near either side of the view (but don't click), hold down the Option key, and use the Left Arrow and Right Arrow keys to nudge all of the settings elements until there is an equal margin between the left end of the leftmost element and the right end of the rightmost element.

Add a little space vertically between the groups. To do this, enlarge the design surface, and then start at the bottom and work upward. First, select the radio group at the bottom. Then move the pointer over the checkbox above it (but don't click), hold down the Option key, and use the Down Arrow key to nudge

the radio group downward until the tag indicates that it is separated from the checkbox above it by 12 pixels. Do the same thing for the two checkboxes above the radio group, selecting all of the groups to be moved and Option-dragging them, and then continue your way upward. I leave 8 pixels between a settings element and the section header above it, and 8 pixels between the header and the settings element above it. When you're done, resize the design surface to be as small as possible while preserving proper margins around all four edges.

The result is a pleasant and easily understood arrangement of user interface elements (**Figure 9.4**).

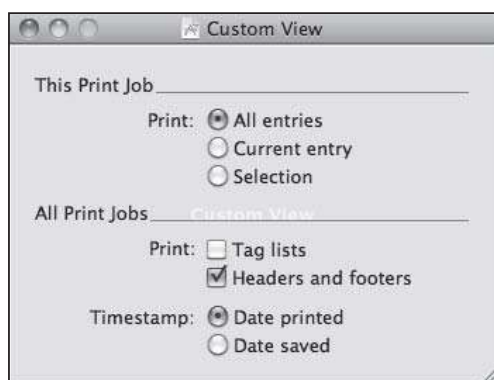


**FIGURE 9.4** The custom accessory view in Interface Builder.

10. The user interface elements in the accessory view don't require help tags because their wording and their labels are clear. However, you should connect the accessibility titles as appropriate. For the radio group on the top, for example, Control-drag from the radio group to its adjacent Print label and select "accessibility title" in the HUD. Repeat the process with each of the other settings elements.

You will return to the nib file in Step 2 to make two additional connections after you have created the accessory view's controller. You will set the view controller to be the file's owner of the nib file, and you will connect the nib file's view outlet to the file's owner.

## Step 2: Create an Accessory View Controller in Xcode

With the accessory view's user interface out of the way, you must next subclass `NSViewController` to create a customized view controller for the accessory view. Name it **DiaryPrintPanelAccessoryController**. In it, you implement accessor methods to get and set the values associated with the user interface elements in the accessory view.

The controller must conform to the `NSPrintPanelAccessorizing` formal protocol by implementing its one required method, `-localizedSummaryItems`, to construct localized



strings for the Print panel's Summary accessory view. If, as here, the accessory view includes features that can affect the appearance of the document in the preview area of the Print panel, you should also implement the one optional method, `-keyPathsForValuesAffectingPreview`.

While writing the view controller, you have a glancing encounter with an important Cocoa concept, *key-value observing* (KVO). The Print panel observes changes in the accessory view's settings and automatically responds by changing the appearance of the preview to match. All you have to do to make this work is to implement the accessor methods and the optional `-keyPathsForValuesAffectingPreview` protocol method. You will learn more about KVO later.

1. In Xcode, choose File > New File. In the New File dialog, select Cocoa Class in the source pane on the left, and then select Objective-C class in the upper pane on the right. Use the pop-up menu to make it a subclass of NSObject. Click Next, enter its filename as **DiaryPrintPanelAccessoryController.m**, select the "Also create 'DiaryPrintPanelAccessoryController.h'" checkbox, and select both the Vermont Recipes and Vermont Recipes SL targets. In the project window, create a new group below the Window Controllers group and name it **View Controllers**; then drag the new accessory view controller files into it.
2. Set the standard file information at the top of the header and implementation files for the DiaryPrintPanelAccessoryController class.
3. In the DiaryPrintPanelAccessoryController.h header file, change the base class from which it inherits from NSObject to NSViewController, and declare that it conforms to the `NSPrintPanelAccessorizing` formal protocol, like this:

```
@interface DiaryPrintPanelAccessoryController :
 NSViewController <NSPrintPanelAccessorizing> {
```

4. Before you forget, go back to the DiaryPrintPanelAccessoryView nib file to set its owner. You may have to build the project first. Select the File's Owner proxy in the nib file's document window, and open the Class pop-up menu at the top of the Object Identity inspector. Look for the DiaryPrintPanelAccessoryController menu item, which now appears near the top of the menu, and choose it to set the file's owner to DiaryPrintPanelAccessoryController.

You must also connect the nib file's view outlet to the Custom View. Select the File's Owner proxy, and open the Diary Print Panel Accessory Controller Connections inspector. Drag from the circle beside the view outlet to the Custom View icon in the nib file's document window.

Save the nib file.

5. Write a convenience or factory method that creates, initializes, and returns a single DiaryPrintPanelAccessoryController instance. In Step 3, you will call this

convenience method from the diary document to create the accessory controller object and add it to the Print panel.

Many classes in the Cocoa frameworks are designed so that only one instantiation can exist at a time, such as `+[NSApplication sharedApplication]`, `+[NSDocumentController sharedDocumentController]`, `+[NSWorkspace sharedWorkspace]`, `+[NSProcessInfo processInfo]`, and `+[NSPrintPanel printPanel]`. The `DiaryPrintPanelAccessoryController` should follow the same model. It is known as a *singleton*, an object that provides a global access point for its methods.

If you were writing a framework as part of a large development team or you intended the framework for wide distribution, you would probably take elaborate steps to make sure that a client of the framework could never accidentally create more than one instance of a singleton at a time or use it in unintended ways. This is called a *strict singleton*. For details about how to do this, read the “Creating a Singleton Instance” section of the *Cocoa Fundamentals Guide*. Here, as the sole developer of the Vermont Recipes application, you are the only person with access to the code and you don’t have to be so paranoid. All you need is a simple method that returns the diary document’s singleton accessory controller, creating it lazily when first needed if it does not yet exist or returning it if it does already exist. You don’t have to worry about accidentally creating another instance, because you’re in charge.

In the `DiaryPrintPanelAccessoryController.h` header file, at the top, declare this method:

```
#pragma mark FACTORY METHOD

+ (DiaryPrintPanelAccessoryController *)sharedController;
```

In the `DiaryPrintPanelAccessoryController.m` implementation file, define it like this:

```
#pragma mark FACTORY METHOD

+ (DiaryPrintPanelAccessoryController *)sharedController {
 static DiaryPrintPanelAccessoryController *sharedController
 = nil;
 if (sharedController == nil) {
 sharedController = [[self alloc] initWithNibName:
 @"DiaryPrintPanelAccessoryView" bundle:nil];
 }
 return sharedController;
}
```

Among the benefits of providing a factory method is encapsulation. The factory method calls `NSViewController`'s designated initializer, `-initWithNibName:bundle:`, passing in the name of the accessory view's associated nib file. In this way, knowledge of the name of the nib file is confined to the `DiaryPrintPanelAccessoryController` class. Otherwise, you would have to use the nib file's name in `DiaryDocument` to instantiate the controller by calling its designated initializer there.

You don't autorelease the controller, as you do with most convenience methods that return objects. There is only one of them, and it remains in existence for reuse throughout the life of the application.

6. Write accessor methods to get and set the values of the user interface elements in the accessory view. You won't declare instance variables for these values, because they reside in a copy of the document's print info object maintained by the accessory view controller.

Each of the accessor methods refers to the view controller's represented object. The use of a *represented object* is a common design pattern in Cocoa. In the case of a print accessory view controller, the *NSViewController Class Reference* explains that when you call `NSPrintPanel`'s `-addAccessoryController:` method, which you will do in Step 3, it automatically calls `NSViewController`'s `-setRepresentedObject:` method to set the controller's represented object to the print info object whose properties will be displayed in the Print panel. This enables you to get the dynamically changing print info object at any time by calling the view controller's `-representedObject` method.

The print info object's settings must be in a form that is compatible with property lists—namely, types such as strings, numbers, dates, Booleans, data objects, and collection types such as dictionaries and arrays. Scalars and structures must be archived in `NSData` objects. Beginning with Leopard, `NSPrintInfo` also has a `-printSettings` method that returns a similar dictionary maintained by the Core Printing system, but Core Printing is an advanced topic beyond the scope of this book.

The print info object maintains a dictionary of current print settings. You access individual settings by calling `NSPrintInfo`'s `-dictionary` method and using the setting's key. Use the key constants defined in `NSPrintInfo`, such as `NSTopMargin` and `NSPrintAllPages`, to get and set built-in print settings. You must define your own keys for your custom accessory view settings in order to include them in the dictionary.

You will refer to the custom print info settings in the `DiaryDocument` class in Step 4, as well as in the `DiaryPrintPanelAccessoryController` class. You should therefore define and declare the keys as external variables, as you did in Recipe 6

with the `VRDefaultDiaryDocumentAliasDataKey` key and in Recipe 7 with the `VRDefaultRecipesWindowStandardSizeKey` key.

There are four print info settings to work with, `printItems`, `printTags`, `print-HeadersAndFooters`, and `printTimeStamp`. I'll present the code for one of them in the book, `printItems`. Look in the downloadable project file for Recipe 9 for the others.

In the `DiaryPrintPanelAccessoryController.h` header file, define them above the `@interface` directive like this example:

```
extern NSString *VRDefaultDiaryPrintItemsKey;
```

Define them at the bottom of the `DiaryPrintPanelAccessoryController.m` implementation file, after the `@end` directive, like this example:

```
NSString *VRDefaultDiaryPrintItemsKey = @"diary print items";
```

Now you can declare and define the accessor methods. The names of these methods include the word *print* as a verb. In the `DiaryPrintPanelAccessoryController.h` header file, declare these example accessor methods:

```
#pragma mark ACCESSOR METHODS
```

- (void)setPrintItems:(NSInteger)option;
- (NSInteger)printItems;

In the `DiaryPrintPanelAccessoryController.m` implementation file, define them like these examples:

```
#pragma mark ACCESSOR METHODS
```

- ```
- (void)setPrintItems:(NSInteger)option {
    [[[self representedObject] dictionary]
        setObject:[NSNumber numberWithInt:option]
        forKey:VRDefaultDiaryPrintItemsKey];
}

- (NSInteger)printItems {
    return [[[[self representedObject] dictionary]
        objectForKey:VRDefaultDiaryPrintItemsKey] integerValue];
}
```

Each of the accessor methods accesses the represented object, which is the accessory view controller's copy of the document's print info object, and each of them gets or sets the print settings using `-dictionary`. The first pair of accessor methods gets and sets an `NSInteger` value where `0` represents the first choice in the radio group, to print all entries, `1` represents the second choice, to print

the current entry, and 2 represents the third choice, to print the selected text. The next two get and set BOOL values. In the last pair, using NSInteger again, 0 represents the first choice in the radio group, to set the timestamp in the footer to the date the document was printed, and NO represents the second choice, to set the timestamp to the date when the document was last saved.

7. Write action methods to invoke the setter methods when the user changes settings in the accessory view. Again, I give one example here. Find the others in the downloadable project file for Recipe 9. In the `DiaryPrintPanelAccessoryController.h` header file, declare them like this example after the accessor methods:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)changePrintItems:(id)sender;
```

Define them at the end of the `DiaryPrintPanelAccessoryController.m` implementation file like this example:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)changePrintItems:(id)sender {
    [self setPrintItems:[sender selectedRow]];
}
```

8. Connect the action methods in Interface Builder. In the `DiaryPrintAccessoryView` nib file, select the radio group at the top of the design surface, and Control-drag to the File's Owner proxy. You could just as well connect the action method to the First Responder proxy, but there is no reason to make use of the responder chain here because the action method is implemented in the file's owner. In fact, NSViewController objects are not even in the responder chain by default, and you would have to splice them in. In the HUD, choose the first action, `changePrintItems:`. A radio group is an NSMatrix object that has only one column of cells, so the `-changePrintItems:` action method calls NSMatrix's `-selectedRow` method to discover which radio button the user selected and forward it to the print info object through the setter method. The `sender` parameter is the radio group matrix.

Connect each of the other three action methods to its checkbox or radio group in the same way. The action methods for the two checkboxes, `-changePrintTags:` and `-changePrintHeadersAndFooters:`, call the sender's `-state` method to determine whether it is `NSOnState` or `NSOffState`. These two constants are defined as NSIntegers, but in the case of a checkbox that does not allow NSMixedState, it is common to interpret them as Boolean values.

The fourth action method, `-changePrintTimeStamp:`, is connected to a radio group having only two radio buttons. You connect it the same way you connected the first radio group.

9. Now write the `NSPrintPanelAccessorizing` protocol methods.

Although the `-keyPathsForValuesAffectingPreview` protocol method is optional, the *NSPrintPanelAccessorizing Protocol Reference* instructs you to implement it if your Print panel includes a print preview and any of your accessory view's settings affect the appearance of the print preview. All four of your settings do affect the print preview, so you must implement this method. Because it is a protocol method declared in the Cocoa frameworks, you need not declare it yourself.

In the `DiaryPrintPanelAccessoryController.m` implementation file, define it like this:

```
#pragma mark PROTOCOL METHODS
```

```
- (NSSet *)keyPathsForValuesAffectingPreview {  
    return [NSSet setWithObjects:@"printItems", @"printTags",  
                                @"printHeadersAndFooters", @"printTimestamp", nil];  
}
```

The *NSPrintPanelAccessorizing Protocol Reference* states that the key paths should all be in the form `@"representedObject.printItems"`, but this is correct only if you are using properties or instance variables without accessor methods. Each string in the method as you have just written it is the name of the accessor method you just wrote to get the value of the indicated print setting. You declared the method in the `DiaryPrintPanelAccessoryController` class, so the method's key path is simply `@"printItems"`. Cocoa uses this key path with Cocoa bindings behind the scenes to monitor changes that the user makes to any of these print settings and to update the print preview accordingly. You will learn about Cocoa bindings later, in Recipe 14.

10. The `-localizedSummaryItems` protocol method is required. It returns an array of localized strings to be displayed in the Summary accessory view in the Print panel. If you have never used the Summary accessory view of a Print panel, look at one in TextEdit or Safari now. It shows all of the accessory views in a collapsed outline view. Expand any topic, and it displays a textual summary of the current values of all of the settings in that accessory view. The Summary accessory view provides a convenient way to examine all of the print settings at once.

It is common to place these localized value strings in a strings file and to read them using Cocoa's `NSLocalizedStringFromTable()` function. You haven't placed a lot of settings into this custom accessory view, however, and it is just as easy to code them in the protocol method and use the `NSLocalizedString()` function. Although the method is long, its logic is very simple. It defines a bunch of local string variables, and then it returns them in an array of dictionaries. The values

placed in the dictionaries depend on the values returned by the accessor methods you just wrote.

Add the `-localizedSummaryItems` protocol method at the end of the `DiaryPrint-PanelAccessoryController.m` implementation file. It is long, so look it up in the downloadable project file for Recipe 9.

Step 3: Add the Accessory View Controller to the Print Panel

You have not yet called `+sharedController`, the convenience method you wrote in Step 2 to load the nib file. In this step, you call it in `DiaryDocument` in the same `-printOperationWithSettings:error:` method that you overrode in Recipe 8 to set up the Save As PDF menu item.

The *NSDocument Class Reference* explains the message flow in a document-based application. When the user chooses the Print command in the application's File menu, the application calls `NSDocument`'s `-printDocument:` action method. The action method in turn calls `NSDocument`'s `-printDocumentWithSettings: showPrintPanel:delegate:didPrintSelector:contextInfo:` method. You can override this or a couple of other methods if you need to write a temporary delegate callback method to do something after printing is complete. This method in turn calls `NSDocument`'s `-printOperationWithSettings:error:` by default, before the application displays the Print panel. The `-printOperationWithSettings:error:` method does nothing by default, and the class reference states that you must override it in order to print. Here is where you create the print operation and its associated Print panel, adding the accessory view controller to the Print panel and loading the accessory view's nib file. Finally, `-printDocument:` calls `NSDocument`'s `-runModalPrintOperation:delegate:didRunSelector:contextInfo:` method to present the Print panel. You will override it in Step 4 to obtain the final values of the custom print settings and save them to the document's copy of the print info object and to the user defaults.

In this step, you focus on overriding `-printOperationWithSettings:error:`, as required in order to create the print operation and configure its Print panel.

1. The first several statements in the `-printOperationWithSettings:error:` method, as you wrote it in Recipe 8 in the `DiaryDocument.m` implementation file for the Save As PDF menu item, are also needed for the print operation. They get the document's print info—every document in a document-based application has one—and then they make an independent copy of it and add the settings passed into the method, if any. Making an independent copy of the

print view here prevents the added values from the `printSettings` argument from polluting the document's print info object. When you call `-printOperationWithView:printInfo:` in a moment, it automatically makes a copy of the combined print info object, again leaving the document's print info object unchanged. This copy is passed to the accessory view controller, but it goes away when the Print panel is dismissed. As a result, the next time the user prints the document, the accessory view's settings revert to their default values. You will arrange later to save some of the changed settings in the user defaults so that they will remain in effect between print jobs unless the user changes them.

Remember that when you implemented the Save As PDF menu item's behavior in Recipe 8, you passed in the setting identifying this as a save job. In the `-saveDocumentAsPDFTo:` action method, you created a dictionary that contained the value `NSPrintSaveJob` under the key `NSPrintJobDisposition`. You passed that dictionary to `-printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:`, and that method passed it along to `-printOperationWithSettings:error:`, where you can now read it. Following the initial statements in the `-printOperationWithSettings:error:` method, which are common to saving and printing the document, you should test the incoming `printSettings` argument to see whether this is a save job. If it is, execute the existing call to `NSPrintOperation`'s `+printOperationWithView:printInfo:` method.

Then, if this is a print job, you add code in an `else` branch to handle it.

To avoid confusion, examine the entire revised method in the downloadable project file for Recipe 9:

```
if ((op == nil) && (outError != NULL)) {
    *outError = [NSError errorWithDomain:NSCocoaErrorDomain
                                code:NSUserCancelledError userInfo:nil];
}
```

For now, the new print job code in the `else` branch uses the existing key diary view in the Chef's Diary window for printing. This is the view you see onscreen, without pagination, headers, footers, or other printing features. You will replace this view with a modified view for printing in Step 5. You need the temporary view here only so that you can test the accessory view at the end of this step.

The new code creates a print operation using the print view. Then it gets the print operation's default Print panel; it adds the accessory view controller to the Print panel, which loads its nib file; and it sets the upper settings area of the Print panel to display some standard user interface elements and omit others.

The proper way to set the Print panel's options is to start with the default options, then add additional options and remove any unwanted options. It is important to start with the default options instead of an empty list because Apple may change the default options in a future version of Mac OS X, and

you don't want your application to fall behind. You add additional options by combining all of them with the default options using the bitwise inclusive OR operator. You remove unwanted options by combining the result with the one's complement of the unwanted option using the bitwise AND operator.

The default options are `NSPrintPanelShowsCopies`, `NSPrintPanelShowsPageRange`, and if you are writing a document-based application, `NSPrintShowsPreview`. The page range control is not suitable for the Chef's Diary, however. Unlike a word processing document, the Chef's Diary is not paginated in its onscreen window, and the user cannot tell in advance which pages to designate for printing. Instead, the custom accessory view allows the user to choose whether to print the entire document, the current entry, or the selected text. You therefore suppress the page range user interface element.

The code you just added creates and initializes a new `DiaryPrintPanelAccessoryController` object, so you must import that class into the `DiaryDocument.m` implementation file. Add this line near the top, with the other `#import` directives:

```
#import "DiaryPrintPanelAccessoryController.h"
```

2. By default, if you do nothing about it, a custom accessory view is identified in the features pop-up menu by the name of the application. This isn't adequate if you add multiple custom accessory views, and it may not be appropriate if you have different custom accessory views for different kinds of documents, as you do in *Vermont Recipes*. For the diary document, it seems a little clearer to title the menu *Vermont Recipes Chef's Diary* in order to distinguish it from the recipes document.

`NSViewController` has `-setTitle:` and `-title` accessor methods to set and get a view controller's title because, according to the *NSViewController Class Reference*, Apple anticipates that `NSViewController` will often be used the way it is used with Print panel accessory views. Apple's advice is that, if you need to give the features pop-up menu in the Print panel a custom title, you should override the `-title` accessor method so that it returns a hard-coded localized value, like this:

```
- (NSString *)title {  
    return NSLocalizedString(@"Vermont Recipes Chef's Diary",  
                             @"title of custom print accessory view for Chef's Diary");  
}
```

My instinct is different. I think the best way to do it is to call `NSViewController`'s `-setTitle:` method to set the title, knowing that the `-title:` method will then return that title. If this were a window controller, you would be accustomed to placing code to do this in the `-windowDidLoad` method, in order to configure the user interface after the nib file is loaded but before the window is displayed. You can't do it exactly that way in an `NSViewController`, however, because it has no

analogous `-viewDidLoad` method. A section in the *Mac OS X SnowLeopard Release Notes: Cocoa Application Framework* helpfully titled “Advice for People Who Are Looking for `-viewWillLoad` and `-viewDidLoad` Methods in `NSViewController`” provides the solution: Simply override `NSViewController`’s `-loadView` method, and place your code just after calling the superclass’s implementation of `-loadView`.

In the `DiaryPrintPanelAccessoryController.m` implementation file, above the Protocol Methods section, override the method like this:

```
#pragma mark OVERRIDE METHODS
```

```
- (void)loadView {  
    [super loadView];  
    [self setTitle:NSLocalizedString(@"Vermont Recipes Chef's Diary",  
        @"title of custom print accessory view for Chef's Diary")];  
}
```

3. Although you aren’t done yet, you are finally able to test your new accessory view to see how it’s coming along. Build and run the application; create a new, empty Chef’s Diary document; enter some text; and choose `File > Print`. After a pause, the Print panel opens. In it, you see that the features pop-up menu is titled `Vermont Recipes Chef’s Diary`, and your custom accessory view is laid out beneath it. The preview shows an image of the document with the text that you just entered (**Figure 9.5**).

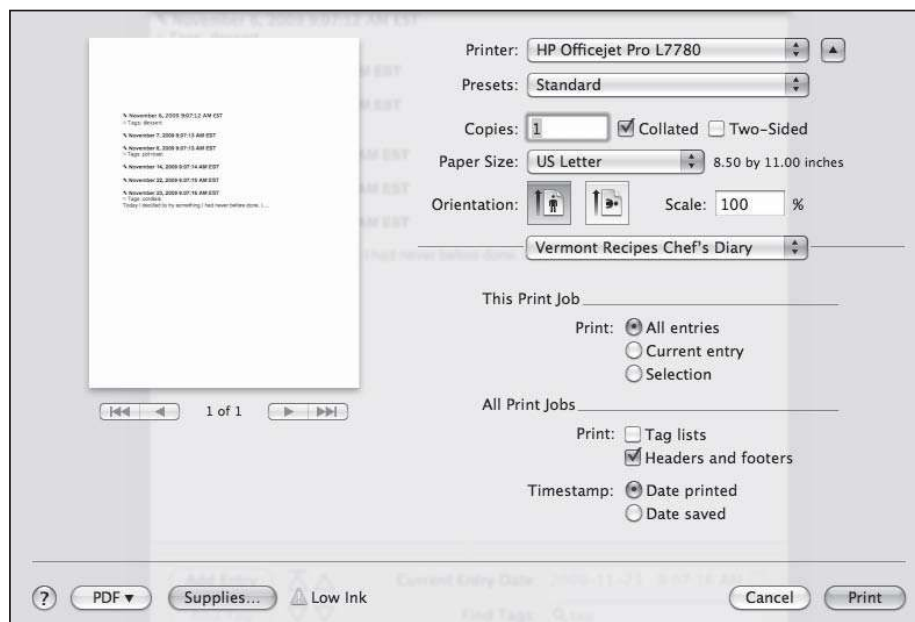


FIGURE 9.5
The Chef’s
Diary Print
panel after
adding custom
features.

The user can, of course, change the settings in the custom accessory view by clicking them. Change all of them now—every single one of them. Then choose `Summary` in the features pop-up menu, and expand the `Vermont Recipes Chef’s Diary` item. The summary correctly reports all of the changed settings (**Figure 9.6**).

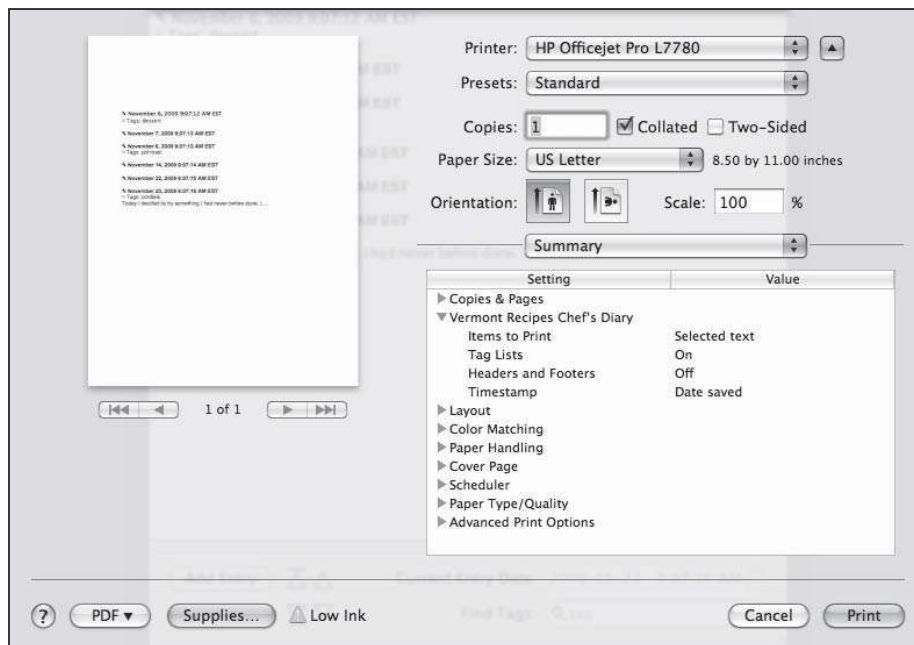


FIGURE 9.6
The Summary accessory view showing changed settings.

Next, click Cancel to close the Print panel, but don't close the document. Reopen the Print panel by again choosing File > Print. You see that all the changes appear still to be in place, judging by the settings of the user interface elements in the accessory view. But there's a problem. Choose Summary from the features pop-up menu, and the summary now does not match the settings of the user interface elements. It is easy, on reflection, to understand why the summary shows empty print info settings instead of the changed settings you see in the accessory view's user interface elements. The view controller's represented object—a copy of the document's print info object—went away when you closed the Print panel, and a new copy was created when you opened the Print panel. All of the changed settings were lost. But the accessory view remembers the previous state of its user interface elements because the Print panel is not released; it sticks around for reuse. The solution is to reset the user interface elements to match the new print info object obtained from the document when the Print panel is loaded. You will fix this problem in this step, but first do some more testing.

Quit the application and relaunch it; create a new, empty Chef's Diary document; and open the Print panel by choosing File > Print. Don't make any changes to the accessory view's settings. You see that the "Headers and footers" checkbox is selected—it contains a checkmark—because that's the way you designed it in Interface Builder in Step 1. Now choose Summary from the features pop-up menu. The Headers and Footers item claims that it is turned off. The problem is that the application and its Chef's Diary document do not know anything about the way you set up the nib file in Step 1. In Step 1, you made the "Headers and footers" checkbox look as if it had been selected only in anticipation of later setting a matching default value in code, and you haven't yet done that. You will implement a comprehensive solution for this issue in Step 4 by arranging to

store custom print settings in the user defaults and to restore them so that, when the user opens the Print panel, they remain in effect. In the meantime, the fix you are about to implement for the first problem will solve this problem, too, by changing the “Headers and footers” checkbox to match the true value of its underlying setting, which is `NSOffState`.

4. To update the accessory view when it loads so that its user interface elements match the document’s print info settings, use standard techniques that you have already seen several times. Create an outlet for each of the four user interface elements in the accessory view, connect them in Interface Builder, and update their state when the accessory view loads.

In the `DiaryPrintPanelAccessoryController.h` header file, declare these instance variables in the `@interface` directive like this example:

```
IBOutlet NSMatrix *printItemsRadioGroup;
```

Declare their getter methods at the top of the Accessor Methods section like this example:

```
- (NSMatrix *)printItemsRadioGroup;
```

Define them at the top of the Accessor Methods section of the `DiaryPrintPanelAccessoryController.m` implementation file like this example:

```
- (NSMatrix *)printItemsRadioGroup {  
    return printItemsRadioGroup;  
}
```

Connect each of the four user interface elements in the `DiaryPrintPanelAccessory-View` nib file by Control-dragging from the File’s Owner proxy to one of the user interface elements and selecting its outlet in the HUD. Save the nib file.

5. Write a method to update the four user interface elements. In the `DiaryPrintPanelAccessoryController.h` header file, declare it like this at the end of the file:

```
#pragma mark VIEW MANAGEMENT
```

```
- (void)updateView;
```

Define it like this at the end of the `DiaryPrintPanelAccessoryController.m` implementation file:

```
#pragma mark VIEW MANAGEMENT
```

```
- (void)updateView {  
    [[self printItemsRadioGroup] setState:NSOnState  
        atRow:[self printItems] column:0];  
}
```



```

[[self printTagsCheckbox] setState:([self printTags])
    ? NSOnState : NSOffState];
[[self printHeadersAndFootersCheckbox] setState:
    ([self printHeadersAndFooters]) ? NSOnState : NSOffState];
[[self printTimestampRadioGroup] setState:NSOnState
    atRow:[self printTimestamp] column:0];
}

```

Each of the statements gets the value of the underlying setting in the print info object by calling its getter method. It then uses that value to set the state of the user interface element to match. In the first statement, for example, to turn on whichever radio button is dictated by the return value of the `-printItems` accessor method, you set it to `NSOnState`. In a radio button group matrix, which can have only one selected button, this automatically turns off any other radio button. The same explanation applies to the last statement. The two methods in the middle are a little simpler, because they just set the checkbox to `NSOnState` or `NSOffState`, depending on the return value of the accessor method.

6. Now comes the hard part: deciding where to call the `-updateView` method. The obvious place would be in the `-loadView` method, just as you set the title of the features pop-up menu in that method. This is where the Snow Leopard AppKit Release Note advises you to place code that should run when a view is loaded.

It will only have the appearance of working, however, and if you don't test this very carefully, you will ship a buggy application. Try it. Place this statement at the end of the `-loadView` method in the `DiaryPrintPanelAccessoryController.m` implementation file:

```
[self updateView];
```

Build and run the application, create a new Chef's Diary document, and choose File > Print. The Print panel opens, and the "Headers and footers" checkbox is deselected. This is exactly what you wanted to see, but you're seeing it here for the wrong reason. To understand the problem, add this statement to the beginning of the `-updateView` method:

```

NSLog(@"in -updateView; represented object exists: %@",
    [self representedObject] ? @"YES" : @"NO");

```

Build and run the application again, create a new Chef's Diary document, and choose File > Print. You see the same result in the accessory view, but in the Debugger Console you now see this log message: "in -updateView; represented object exists: NO." The represented object is supposed to be a copy of the document's print info object, but the log message says it doesn't exist.

It turns out that the Cocoa printing system sets the accessory view's represented object *after* loading the view. When you called `-updateView` in `-loadView`, the

accessory view didn't yet have any information about the state of the settings in the document's print info object. The `-updateView` method properly deselected the "Headers and footers" checkbox when the `-printHeadersAndFooters` accessor method returned `NO`. However, it did so for the wrong reason. The accessor method returned `NO` only because the represented object and its dictionary were `nil`. Cocoa allows you to send messages to a `nil` object, but in a case like this, it returns a spurious `NO` result. If the document's print info object happened to contain a value of `YES` for the headers and footers setting, the application would still have deselected the checkbox, which would be wrong.

You have two choices for fixing this problem. The cleverest solution is unacceptable because it causes the checkbox to flicker. Try it to see what I mean. Edit the statement in `-loadView` that calls `-updateView` so that it now calls `-updateView` after a delay, like this:

```
[self performSelector:@selector(updateView)
      withObject:nil afterDelay:0.0];
```

Every Cocoa object descended from `NSObject` responds to the `-performSelector:withObject:afterDelay:` method. Sometimes it can be quite useful, allowing you to specify a fractional number of seconds to delay—or even to delay for many seconds—before the method is executed. It has an even more important use, however, which you see here. When you specify `0.0` as the delay period, as you did here, it does not delay for a set period of time, but only until the next iteration of the run loop. That is, it allows Cocoa to finish doing everything it is supposed to do in the current iteration of the run loop, and then immediately executes the delayed statement on the next iteration.

Although Cocoa calls `-setRepresentedObject:` after it loads the accessory view controller, it does so in the same iteration of the run loop. By delaying the call to `-updateView` to the next iteration, you allow the accessory view controller to acquire knowledge of the represented print info object. When you build and run the application and perform the experiment again, the Debugger Console reports, "in `-updateView`; represented object exists: YES." The checkbox is again deselected, but this time it's for the right reason.

Unfortunately, in addition to calling `-setRepresentedObject:` in the same iteration of the run loop, Cocoa also shows the Print panel in the same iteration. Thus, the first time you open the Print panel after launching the application, you see the checkbox and its checkmark for a split second as the Print panel opens, and then you see the checkmark disappear. This is not a good design.

The only acceptable solution, therefore, is to override the `-setRepresentedObject:` method itself and place the call to `-updateView` in it, immediately after calling the superclass's implementation of `-setRepresentedObject:`. This is essentially

what TextEdit does, but it took this roundabout route for me to figure out why. Now the user interface elements in the accessory view are updated after the accessory view's represented print info object is set but before the print panel is displayed.

You have solved the first problem. The accessory view now updates its user interface elements to match the document's print info settings. However, you wanted the "Headers and footers" checkbox to be turned on by default, and it isn't. You will devote the next step to fixing that problem, and at the same time, you will devise a way to save some of the user's changed print settings in the user defaults so that they persist the next time the user runs the application.

Step 4: Save Custom Print Settings

In Step 1, you designed the accessory view in Interface Builder so that it has two sections, one that applies only to the current print job and one that applies to all print jobs. This means that the first setting is supposed to return to its initial default value every time you close and reopen the print panel, including when you quit the application and relaunch it. The settings in the second section, to the contrary, are supposed to persist through closing and reopening the print panel, closing and reopening the Chef's Diary document, and even quitting and relaunching the application.

The *HIG* and other Apple documents relating to printing give developers great latitude in deciding upon the appropriate persistence of print settings. For example, Apple advises that document-based applications can keep print settings alive on a per-document basis as long as the document is open or even after the document is closed and reopened. You can see this flexibility in the sections of the print panel controlled by the printing system, too. For example, in TextEdit, you can change virtually all of the settings in the top part of the print panel, and they return to their defaults the next time you open the print panel, whether you reopen it after clicking Cancel, after clicking Print, or after choosing Print to PDF from the PDF pop-up menu. Yet if you change the "Print header and footer" setting in the TextEdit accessory view, the change persists whether you reopen the print panel after clicking Cancel, clicking Print, or choosing Print to PDF from the PDF pop-up menu. It's a matter of your best judgment about the typical usage of your application and what your users will find most convenient.

In Vermont Recipes, my judgment is that users expect their choice about whether to print the entire document, the current entry, or the current selection to return to the default "All entries" setting every time they reopen the Print panel. This section replaces the standard page range setting, which behaves the same way. I judge that users will prefer that the remaining accessory view settings persist across multiple printings of the Chef's Diary. They will usually, I believe, want to omit tag lists,

include headers and footers, and have the date item in the footer represent the date the document was printed instead of the date it was last saved. The Chef's Diary is a one-of-a-kind document, which makes it appropriate to keep these settings consistent across multiple printings of the one document.

In a typical document-based application, you would save each document's print settings on disk in the document itself, based on the settings in the Page Setup dialog. The Chef's Diary is a one-of-a-kind document, however, and it is more convenient to save its print settings in the application's user defaults. They will of course be separate from whatever settings you design later for the recipes document.

You have used the user defaults a number of times already, but you have used them only to save and retrieve values on the fly. The print settings for the Chef's Diary are a little different in that one of them, the headers and footers setting, is supposed to have an initial default value of YES even before the user is ready to print the first copy of the document. There is a standard technique for setting up initial default values in the user defaults database, which you use in this step. You set initial default values in an `+initialize` method that is executed before the document is created. Thereafter, any different value set by the user takes the place of the initial default value.

The user defaults settings for printing the Chef's Diary, like the print info object, are associated with the document, not with the Print panel accessory view. The document implements the `+initialize` method to set up the initial default values. The document loads the current user defaults settings into its print info object when the user creates or opens the document. The document saves them back to the user defaults every time the user closes the Print panel, whether or not the user saves the document. The document may also synchronize the user defaults to disk from time to time while the document is open.

The accessory view controller knows nothing at all about the user defaults. All it knows about the print settings is that they are in its represented object, which was set to an independent copy of the document's print info object when the user opened the Print panel. The accessory view controller records any changes the user makes to these settings in the represented object, using the setter methods you wrote in Step 2. When the user closes the Print panel, its copy of the print settings would go away and all the changes the user made would be lost, unless you arrange to save the ones that should persist.

To save the settings that the document needs to remember, you override one of `NSDocument`'s printing methods, `-runModalPrintOperation:delegate:didRunSelector:contextInfo:`, and save the print operation object in its `contextInfo` parameter. When the user closes the Print panel, the temporary delegate callback method runs, getting the saved print operation from the `contextInfo` argument. The callback method grabs the changed custom print settings from the saved print

operation's dictionary. It uses them to update the document's own copy of the print info object, ready for the next time the user opens the Print panel. At the same time, it saves these changed custom print settings to the user defaults, as described earlier.

The technique described here differs from the way many developers implement printing and even from some of Apple's code examples. Many developers implement an accessory view controller's accessor methods to save changed values directly to the user defaults and retrieve them directly from the user defaults. However, this bypasses the careful separation that the Cocoa printing system maintains between the print info object in the document and the copy of the print info object in the accessory view.

One of the advantages of the technique you apply here is that it maintains encapsulation: The accessory view and its controller know only about the copy of the print settings object that they were given; they know nothing about the document's copy or even about the user defaults settings. The document handles the relevant user defaults, just as it maintains control over its own print info object. In addition, it isolates in the document the decision as to which custom settings should persist and which should not. Other advantages are that this technique leaves your application free to take advantage of more advanced features of the printing system and future developments.

1. Start by registering the initial default values of your custom print settings in the user defaults. You don't care about the value of the print items setting, which the user sets using the radio group at the top of the custom accessory view, and you don't save it to user defaults. It automatically defaults to its first setting, "All entries," every time the user opens the Print panel. Initial default values of `nil` or `0` never have to be set up anyway, and the "All entries" setting is `0`.

The other three custom print settings should persist across print jobs, so they should be saved in the user defaults.

At the top of the `DiaryDocument.m` implementation file, as the first method in the existing Initialization section, add this class method:

```
+ (void)initialize {
    if (self == [DiaryDocument class]) {
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        NSDictionary *initialUserDefaults =
            [NSDictionary dictionaryWithObjectsAndKeys:
             [NSNumber numberWithInt:NO], VRDefaultDiaryPrintTagsKey,
             [NSNumber numberWithInt:YES],
             VRDefaultDiaryPrintHeadersAndFootersKey,
             [NSNumber numberWithInt:0],
             VRDefaultDiaryPrintTimestampKey, nil];
        [defaults registerDefaults:initialUserDefaults];
    }
}
```


The `+initialize` method is declared in `NSObject`. You should read the description of the `+initialize` method in the *NSObject Class Reference* now, because it explains exactly how it works and why you start with the test in the first line. In summary, the Cocoa runtime calls the `+initialize` method of every such class, if the class implements it, once before calling any other method on the class. It is therefore the perfect place to put code that must execute very early in the life of an object.

The test in the first line is necessary, because it is possible for the runtime to call the `+initialize` method of the class more than once. Specifically, if a subclass of the class does not implement `+initialize`, the runtime's call to the subclass's `+initialize` method falls through to the superclass's implementation. You can't use `[self isKindOfClass:[DiaryDocument class]]` or `[self isKindOfClass:[DiaryDocument class]]` to avoid running the body when this happens. Those tests would ask whether `self` is an *instance* of the `DiaryDocument` class or, in the first statement, an *instance* of a subclass, but `+initialize` is a class method and `self` is the class object. Objective-C has the notion of a *class object*, a real object that can execute class methods but can't have instance variables or execute instance methods. It is sometimes called a *factory object* because one of its uses is to create instances that can have instance variables and respond to instance methods. When the `+initialize` method is called the first time on `DiaryDocument`, `self` is the `DiaryDocument` class object and the test evaluates to true. If it is called subsequently, `self` is still a class object, but of a subclass of the class, and the test evaluates to false. You can read about class objects in the "Class Objects" section of *The Objective-C Programming Language*.

The rest of the code in the `+initialize` method sets up the initial default values for the three print settings in the user defaults. First, it creates a local `NSDictionary`, `initialUserDefaults`, with three of the print settings keys you declared and defined in `DiaryPrintPanelAccessoryController` in Step 2. It is convenient to use the same keys both for the print info dictionary and the user defaults. You can set `BOOL` values in the user defaults by using the `NSString` values `@YES` and `@NO`, but here you need `NSNumber` objects because they will be used in the print info object, as well as in the user defaults.

The `+initialize` method then registers this local dictionary by calling the `NSUserDefaults` method `-registerDefaults:`. This does not write the initial defaults to disk. Instead, it executes every time the application runs, setting up these defaults in memory for temporary use until the user quits. These initial defaults are registered in the registration domain maintained by the user defaults, which has the lowest priority and is superseded by the defaults you save later to the application domain. You first save values to the application domain after the user sets different values and then closes the Print panel. Once that happens, the user defaults system uses the application domain settings from then on and ignores the initial defaults set up by the `+initialize` method.

2. Next, modify the document's print info object by adding these settings to it. Do this in the DiaryDocument's designated initializer, `-init`, so that the print info object will have the benefit of the initial default values as soon as the document opens. When the user chooses File > Print, the code you have already written picks up this modified version of the print info object containing these initial default values, including YES for the print headers and footers setting. Shortly, you will add user-altered values to the user defaults if any changes were made during a print operation, and your code in the `-init` method will pick them up instead.

In the DiaryDocument.m implementation file, after the new `+initialize` method, add this method:

```
- (id)init {
    if ((self = [super init])) {
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        NSDictionary *customPrintSettings =
            [NSDictionary dictionaryWithObjectsAndKeys:
             [defaults objectForKey:VRDefaultDiaryPrintTagsKey],
             VRDefaultDiaryPrintTagsKey,
             [defaults objectForKey:
              VRDefaultDiaryPrintHeadersAndFootersKey],
             VRDefaultDiaryPrintHeadersAndFootersKey,
             [defaults objectForKey:VRDefaultDiaryPrintTimestampKey],
             VRDefaultDiaryPrintTimestampKey, nil];
        [[[self printInfo] dictionary]
         addEntriesFromDictionary:customPrintSettings];
    }
    return self;
}
```

If you were to build and run the application now, you could create a new Chef's Diary document and choose File > Print, and the Print panel would now show the "Headers and footers" checkbox selected. The Summary view would also show that it was turned on.

3. When the user closes the Print panel, the document should replace the values of these three settings in the document's existing print info object with any new values in the accessory view controller's copy of the print info object, because they are supposed to be persistent. Don't bother checking whether the settings were in fact changed. The operation is fast enough that you should just perform the substitution anyway.

How can the document gain access to the accessory view controller's copy of the print info object when the Print panel is closed? The answers to questions

like this can almost always be found by reading the relevant class references looking for methods to override. You have already seen several examples of this—for example, when you examined the message flow diagrams in the *Document-Based Applications Overview* in previous recipes. When you traced out the message flow in the `-printDocument:` action method earlier in this step, you found that Cocoa automatically calls `-runModalPrintOperation:delegate:didRunSelector:contextInfo:`. This is the logical candidate because it has access to the print operation object, and it allows you to run a temporary delegate callback method after the print operation completes and the Print panel closes.

You can save the print operation that runs the Print panel in the method's `contextInfo` argument, and you can get it back in the `contextInfo` argument of the callback method after the Print panel closes. This print operation contains the copy of the print info object that is used by the Print panel, where all the user changes are recorded. It therefore gives you a convenient way to access the user's changes after the Print panel is closed.

In the `DiaryDocument.m` implementation file, at the end of the Override Methods section, override the method:

```
- (void)runModalPrintOperation:(NSPrintOperation *)printOperation
    delegate:(id)delegate didRunSelector:(SEL)didRunSelector
    contextInfo:(void *)contextInfo {
    [super runModalPrintOperation:printOperation delegate:self
        didRunSelector:@selector
            (diaryDocumentDidRunModalPrintOperation:success:contextInfo:)
        contextInfo:[printOperation retain]];
}
```

Your override method simply calls the superclass's implementation, designating `self`—the document—as temporary delegate and specifying the callback method you will write in a moment. The key point here is that the method you are overriding hands you the print operation object in its first argument, and you then save it in the `contextInfo` argument of your call to the superclass's implementation. You retain the print Operation object because you don't want it to go away when the Print panel closes. The print operation's dictionary changes as the user changes your custom settings in the Print panel's accessory view, and those changes remain available to you when you examine the print operation's dictionary returned to you in the callback method's `contextInfo` argument.

Note that this implementation of `-runModalPrintOperation:delegate:didRunSelector:contextInfo:` discards any `delegate` and `contextInfo` argument that might have been passed in. This works here because you know that they are `NULL`. For general use, it would be important to capture them and do something with them.

Now insert the callback method immediately below the method you just wrote:

```
-(void)diaryDocumentDidRunModalPrintOperation:(NSDocument *)document
    success:(BOOL)success contextInfo:(void *)contextInfo {
    NSPrintOperation *printOperation = contextInfo;
    NSMutableDictionary *changedDictionary =
        [[printOperation printInfo] dictionary];
    NSDictionary *customPrintSettings =
        [NSDictionary dictionaryWithObjectsAndKeys:
        [changedDictionary objectForKey:VRDefaultDiaryPrintTagsKey],
        VRDefaultDiaryPrintTagsKey,
        [changedDictionary objectForKey:
        VRDefaultDiaryPrintHeadersAndFootersKey],
        VRDefaultDiaryPrintHeadersAndFootersKey,
        [changedDictionary objectForKey:
        VRDefaultDiaryPrintTimestampKey],
        VRDefaultDiaryPrintTimestampKey, nil];
    [[[self printInfo] dictionary]
        addEntriesFromDictionary:customPrintSettings];

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [defaults setObject:[changedDictionary
        objectForKey:VRDefaultDiaryPrintTagsKey]
        forKey:VRDefaultDiaryPrintTagsKey];
    [defaults setObject:[changedDictionary
        objectForKey:VRDefaultDiaryPrintHeadersAndFootersKey]
        forKey:VRDefaultDiaryPrintHeadersAndFootersKey];
    [defaults setObject:[changedDictionary
        objectForKey:VRDefaultDiaryPrintTimestampKey]
        forKey:VRDefaultDiaryPrintTimestampKey];

    [printOperation release];
}
```

The first block of statements takes the incoming `contextInfo` argument, which is the print operation that the Print panel just finished with; extracts the dictionary of print settings from the print operation's `printInfo` object; turns them into a temporary local dictionary; and adds the dictionary's entries to the document's `printInfo` object. If the user reopens the Print panel while the document is still open, the document will send a copy of this modified print info object to the Print panel, and these custom print settings will be reflected in the accessory view's user interface elements and in the Summary view.

The second block of statements immediately takes the same changed values and saves them to the user defaults. The next time the user closes the diary document and reopens it to print it again, or quits the application and relaunches it to print the diary document again, these custom print settings will again be reflected in the accessory view's user interface elements and in the Summary view.

You could reduce the amount of code by consolidating all of the settings of interest in an array and maintaining it using aggregate operations like `-dictionaryWithValuesForKeys:`. Here, the more verbose approach is used to make clear what is happening.

4. There remain a couple of cosmetic loose ends to tie up. For one thing, you are no longer using the Page Setup menu item, so get rid of it. Rather than simply deleting it from the MainMenu nib file, however, you should disable it while the diary document is active. You don't yet know whether you will need it for the recipes document.

A quick look at the MainMenu nib file reveals that the Page Setup menu item in the File menu is connected to the `-runPageLayout:` action method in NSDocument. It must be validated in DiaryDocument when DiaryDocument is active and in the responder chain. In the existing `-validateUserInterfaceItem:` method in the DiaryDocument.m implementation file, add this branch to the chained `if` tests:

```
} else if (action == @selector(runPageLayout:)) {  
    return NO;
```

Now the Page Setup menu item is disabled whenever the diary document is active.

5. The other cosmetic glitch is really two related glitches: In the accessory view, the "Current entry" radio button is enabled even when there is no current entry in the diary document, and the Selection radio button is enabled even when there is no selected text in the diary document. These radio buttons should be disabled as appropriate when there is no current entry or no selection to print.

This is not a job for a `-validateUserInterfaceItem:` method. These radio buttons need not be validated interactively. The Print panel is a document-modal sheet, so the user cannot change the selection, move the insertion point, or add entries to the document while the Print panel is open. You only need to validate these radio buttons a moment before the Print panel is displayed. You can therefore do it in the `-updateView` method you wrote in this recipe, which is called from the `-setRepresentedObject:` method in the DiaryPrintPanelAccessoryController.m implementation file. Don't call it from the `-loadView` method, because it isn't called every time the Print panel opens.

The only challenge is to figure out how to gain access to the diary document and the diary window controller in order to determine the current selection and insertion point and whether the insertion point is in a diary entry. This turns out to be

easy. `NSDocumentController` has a class method, `+sharedDocumentController`, which you have used before. It is available globally, including inside the accessory view controller. It includes a method to get the application's active document, `-currentDocument`, and with that you can call `-windowControllers` to get the document's first and only window controller. Since you validate these Print panel controls only when the sheet is opened, `-currentDocument` is guaranteed to be the diary document. Alternatively, since `DiaryPrintPanelAccessoryController` inherits from `NSViewController`, you could call `[[[self view] window] windowController] document`.

The rest is easy, as you use methods in `DiaryDocument` and `DiaryWindowController` that you have already written. Add these statements to the end of the `-updateView` method:

```
DiaryDocument *document =
    [[NSDocumentController sharedDocumentController] currentDocument];
DiaryWindowController *windowController =
    [[document windowControllers] objectAtIndex:0];
NSUInteger insertionPointIndex = [windowController insertionPointIndex];
BOOL isCurrentEntry = [document currentEntryTitleRangeForIndex:
    insertionPointIndex].location != NSNotFound;
[[[self printItemsRadioGroup] cellAtRow:1 column:0]
    setEnabled:isCurrentEntry];
NSArray *selection = [[windowController keyDiaryView] selectedRanges];
BOOL isSelection = (([selection count] > 1)
    || ([[selection objectAtIndex:0] rangeValue].length > 0));
[[[self printItemsRadioGroup] cellAtRow:2 column:0]
    setEnabled:isSelection];
```

6. Import the `DiaryDocument` and `DiaryWindowController` headers, since the code you just added calls methods from both. Add these statements above the `@implementation` directive in the `DiaryPrintPanelAccessoryController.m` implementation file:

```
#import "DiaryDocument.h"
#import "DiaryWindowController.h"
```

7. You are now through with the nuts and bolts of displaying the accessory view and capturing its changing print settings. Build and run the application to test it. Create a new Chef's Diary document, type some text into it, choose `File > Print`, change all of the custom print settings, click `Cancel`, and quit the application. Then repeat the chain of operations to the point where you open the Print panel. You see that all of the settings in the bottom section of the custom accessory view have been restored to the changed values you set in the first run.

In the next step, you finally get to the point of this recipe: to print a view that reflects the print settings you have constructed.

Step 5: Create a Print View to Print the Document's Content

There are three basic ways to print a multipage document in Cocoa: automatic pagination, customized automatic pagination, and fully customized pagination. The automatic pagination approaches are easy to implement because Cocoa does most of the work, but they are most useful when your document content has fixed page sizes and you aren't interested in adding custom features. Customized automatic pagination does allow you to make some changes—for example, by adjusting page breaks to avoid splitting printed subviews across page boundaries. Here, you will fully customize the pagination of the Chef's Diary.

In Step 3, you implemented the `-printOperationWithSettings:error:` method to print the diary document after the user chooses `File > Print`. To permit preliminary testing of the custom accessory view, you used the existing onscreen document view as a temporary print view. You would be done now if the onscreen view already contained all the features you want to print, such as headers and footers and pagination. TextEdit does this, for example. Apple's printing documentation encourages developers to implement formatting in the document and to confine the print settings to print options such as paper size. For the best user experience, according to the documentation, the appearance of the document onscreen and in print should typically be identical. If you do it this way, printing is almost automatic, requiring very little additional code.

The Chef's Diary, however, is not a full-fledged word processing document, and its onscreen view does not incorporate headers and footers or pagination. These features will be confined to the printed version of the document. Think of the onscreen version as a draft and the printed version as the final copy. For a future release of Vermont Recipes, you might consider enhancing the onscreen display of the document, but for now focus on pagination, headers and footers and the like when printing to paper.

The final step in adding full printing capability to the diary document, therefore, is to create a custom print view and use it instead of the onscreen view. The print view is responsible for constructing a special version of the document's content that honors the user's choice of content settings in the Print panel, such as printing the entire document or only the current entry or selection, and omitting tag lists. It is also responsible for providing additional material, such as headers and footers. To customize printing in this way, the print view must override these three methods of `NSView`: `-knowsPageRange:`, `-rectForPage:`, and `-drawRect:`. There are several other methods that the print view can override optionally. Name the custom print view **DiaryPrintView**.

1. To get it out of the way at the beginning, revise the `-printOperationWithSettings:error:` method now so that it instantiates and initializes the new print view. The method is in the `DiaryDocument.m` implementation file. Replace the existing statement that sets the `printView` local variable to the window controller's key view with this statement:

```
NSArray *printView =  
    [[[DiaryPrintView alloc] initWithFrame:NSZeroRect] autorelease];
```

The `-initWithFrame:` method is the designated initializer for `NSView`. You will reset the frame size in the `DiaryPrintView` object shortly, based on the user's settings in the Print panel.

You must also import the `DiaryPrintView` header into `DiaryDocument` by adding this line near the top of the `DiaryDocument.m` implementation file:

```
#import "DiaryPrintView.h"
```

2. Now create the new `DiaryPrintView` class. Choose `File > New File`. In the New File dialog, select `Cocoa Class` in the source view, select "Objective-C class" in the pane to the right, choose `NSView` from the "Subclass of" pop-up menu, and click Next. Name the file **DiaryPrintView.m**, and select the "Also create 'DiaryPrintView.h'" checkbox. Make sure that both the Vermont Recipes and Vermont Recipes SL target checkboxes are selected, and click Finish. In the Xcode project window, drag the new header and implementation files into the View & Responders group you previously created in the Groups & Files pane. Then revise the file information at the top of both files in the format you have been using all along. Notice that the implementation file already provides stubs for the `-initWithFrame:` and `-drawRect:` methods.
3. In the course of writing the `DiaryPrintView` class's methods, you are going to need four instance variables and their associated accessor methods. Write them now so that this housekeeping task doesn't distract you while you're writing the methods that do the work of printing the document.

Declare the instance variables in the `DiaryPrintView.h` header file, within the curly braces of the `@interface` directive, like this:

```
DiaryDocument *currentDocument;  
NSDate *currentDate;  
NSPrintOperation *currentOperation;  
NSTextStorage *textStorage;
```

Since the class of one of these instance variables, `currentDocument`, is a class you wrote as part of the Vermont Recipes project, you must also import the header

in which that class is declared. At the top of the DiaryPrintView.h header file, just under `#import <Cocoa/Cocoa.h>`, add this:

```
#import "DiaryDocument.h"
```

Declare the accessor methods at the top of the class declaration in the header file like these examples, looking in the downloadable project file for Recipe 9 for the other three:

```
#pragma mark ACCESSOR METHODS
```

- (void)setCurrentDocument:(DiaryDocument *)document;
- (DiaryDocument *)currentDocument;

Define them using the recommended technique for most object accessors, in the DiaryPrintView.m implementation file, like these examples:

```
#pragma mark ACCESSOR METHODS
```

- (void)setCurrentDocument:(DiaryDocument *)document{
 if (currentDocument != document) {
 [currentDocument release];
 currentDocument = [document retain];
 }
}
- (DiaryDocument *)currentDocument {
 return [[currentDocument retain] autorelease];
}

Two of the instance variables, `currentDocument` and `currentDate`, can be initialized when the DiaryPrintView itself is created and initialized. When you created DiaryPrintView, the template provided a stub implementation of its designated initializer, `-initWithFrame:.` Fill in the stub method now so that it looks like this:

- (id)initWithFrame:(CGRect)frame {
 self = [super initWithFrame:frame];
 if (self) {
 currentDocument = [[[NSDocumentController
 sharedDocumentController] currentDocument] retain];
 currentDate = [[NSDate date] retain];
 }
 return self;
}

You will instantiate and initialize the other two instance variables, `currentOperation` and `textStorage`, later. However, you should arrange now to deallocate all of them once the document has been printed. Insert this method after the designated initializer in the `DiaryPrintView.m` implementation file:

```
- (void)dealloc {
    [textStorage release];
    [currentOperation release];
    [currentDate release];
    [currentDocument release];
    [super dealloc];
}
```

4. With the foundation laid, get the diary document's contents and massage them for printing. This isn't the first method to be called during the printing operation, but it is the basis for understanding how the printable content is constructed. The print view constructs a copy of the document's contents for printing based on the user's content choices in the Print panel. In Vermont Recipes, the user can choose whether to include the entire document, the current entry, or the current selection, and whether to omit the tag lists. It is convenient to construct the content to be printed in a separate method if it involves any complexity. Name the method `-printableContent`.

Documents that are not text documents follow a similar strategy, constructing an object or multiple objects for printing based on any settings in the Print panel that affect content. A database document might construct a bunch of carefully arranged text fields for printing based on search or filter criteria, and a spreadsheet document might construct a collection of rows and columns. How you write a print view is always dependent on the nature of the document's contents. Nevertheless, this example using a text storage object for printing illustrates the process.

The diary document is a Rich Text Format (RTF) file, so you take advantage of the very powerful text manipulation features of the Cocoa text system to assemble the content to print. Your basic strategy for printing the diary document is to create a text storage object and copy the diary document's text storage into it, and then to modify the copy. Recall that `NSTextStorage` is a subclass of `NSMutableAttributedString`. The `-printableContent` method you are about to write therefore returns an `NSTextStorage` object. The print view maintains its copy in an instance variable because several of the print view's methods need it.

When you created the print view at the beginning of this step, you based it on `NSView`. If you had based it on `NSTextView`, you would have ended up with a bunch of Cocoa text system objects automatically. Although you can do it that way, you don't need all of the baggage that comes with an `NSTextView` for

printing. In fact, you don't need an `NSTextView` at all, but only an `NSTextStorage` object, an `NSLayoutManager` object, and one `NSTextContainer` object for each page. It is therefore simpler to create the print view by subclassing `NSView` and creating your own hierarchy of text system objects.

Write the `-printableContent` method now. You will arrange to call it and assign its return value to an instance variable shortly, after you have also paginated the content.

In the `DiaryPrintView.h` header file, declare the method at the end like this:

```
#pragma mark UTILITY METHODS
```

```
- (NSTextStorage *)printableContent;
```

Write the implementation code in two steps, starting with the code that makes the fundamental choice of whether to print the entire document, the current entry, or the current selection. This code is controlled by the setting for the `VRDefaultDiaryPrintItemsKey` key. You will add the code for the `VRDefaultDiaryPrintTagsKey` key afterward.

In the `DiaryPrintView.m` implementation file, define the method like this:

```
#pragma mark UTILITY METHODS
```

```
- (NSTextStorage *)printableContent {
    NSTextStorage *storage = [[NSTextStorage alloc] init];
    DiaryDocument *document = [self currentDocument];
    DiaryWindowController *windowController =
        [[document windowControllers] objectAtIndex:0];
    NSDictionary *printInfoDictionary =
        [[[self currentOperation] printInfo] dictionary];

    NSInteger printItems = [[printInfoDictionary
        objectForKey:VRDefaultDiaryPrintItemsKey] integerValue];
    switch (printItems) {
        case 0:
            [storage setAttributedString:[document diaryDocTextStorage]];
            break;
        case 1: {
            NSUInteger insertionPointIndex =
                [windowController insertionPointIndex];
            NSUInteger startLocation = [document
                currentEntryTitleRangeForIndex:
                insertionPointIndex].location;
```

```

        NSInteger endLocation = [document
            nextEntryTitleRangeForIndex:
                insertionPointIndex].location;
        if (endLocation == NSNotFound)
            endLocation = [[document diaryDocTextStorage] length];
        [storage setAttributedString:[document diaryDocTextStorage]
            attributedSubstringFromRange:NSMakeRange(startLocation,
                endLocation - startLocation)]];
        break;
    }
    case 2: {
        NSRange firstSelectionRange = [[[viewController
            keyDiaryView] selectedRanges] objectAtIndex:0]
            rangeValue];
        [storage setAttributedString:[document diaryDocTextStorage]
            attributedSubstringFromRange:firstSelectionRange]];
        break;
    }
}
}

```

The method begins by creating and initializing a text storage object and assigning it to a local variable. Then it sets local variables for the document, its window controller, and the print info dictionary. It gets the document from the `currentDocument` instance variable you initialized in the `DiaryPrintView` class's designated initializer a moment ago. It gets the current print operation, which contains the print info dictionary, from the `currentOperation` instance variable. You will write the code that initializes it in a moment.

You learned in Step 2 that the print info dictionary is a key part of the Cocoa printing system, holding all of the Print panel's settings, including custom settings. Here, you get the setting for the custom `VRDefaultDiaryPrintItemsKey` key, which you set up in Step 2. It is an unsigned integer where 0 represents the entire document, 1 represents the current entry, and 2 represents the current selection. A `switch` statement branches to the appropriate case.

To print the entire document content, case 0 simply calls `-setAttributedString:` to assign the diary document's `diaryDocTextStorage` to the print view's own `textStorage`. The `diaryDocTextStorage` object is an `NSTextStorage` object and therefore a form of mutable attributed string.

To print the current entry, case 1 must do a little more work. It computes the range of the current entry, including its title, any tag list, and its text. Then it uses this range to extract the current entry from the `DiaryDocTextStorage` as

a substring, which it assigns to the print view's text storage. To do this, it calls the `-currentEntryTitleRangeForIndex:` and `-nextEntryTitleRangeForIndex:` methods that you wrote in Recipe 4, passing the `DiaryWindowController`'s `-insertionPointIndex` to each, to get the location of the start and end of the current entry, handling the special case where there is no next entry but only the end of the file. Simple subtraction yields the range of the current entry.

To print the current selection, case 2 gets the `DiaryWindowController`'s `-keyDiaryView` and assigns the text in its selection range to the print view's text storage. The `-selectedRanges` method is guaranteed to contain no empty ranges unless there is only a single range, and if the first and only range is empty, nothing will be printed. Snow Leopard includes a new facility to print the current selection, but you roll your own here.

To complete the work of the `-printableContent` method, add the following section at the end. It removes all tag lists from the text storage that it constructed in the first section, if the setting for the `VRDefaultDiaryPrintTagsKey` key is YES, and then returns the finished printable text storage object.

```

    BOOL printTags = [[printInfoDictionary
        objectForKey:VRDefaultDiaryPrintTagsKey] boolValue];
    if (!printTags) {
#ifdef MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_6
        NSMutableArray *removeRangeArray = [NSMutableArray array];
        NSRange tagRange = [DiaryDocument
            firstTagRangeInString:[storage string]];
        if (tagRange.location != NSNotFound) {
            while (tagRange.location != NSNotFound) {
                [removeRangeArray addObject:
                    [NSValue valueWithRange:tagRange]];
                tagRange = [DiaryDocument nextTagRangeForIndex:
                    NSMaxRange(tagRange) - 1 inString:[storage string]];
            }

            for (NSInteger idx =
                [removeRangeArray count] - 1; idx >= 0; idx--) {
                NSRange substringRange =
                    [[removeRangeArray objectAtIndex:idx] rangeValue];
                [storage deleteCharactersInRange:[storage string]
                    paragraphRangeForRange:substringRange];
            }
        }
    }
    #else

```



```

        [[storage string] enumerateSubstringsInRange:
            NSRange(0, [storage length])
            options:NSMakeRangeEnumerationByParagraphs
            | NSStringEnumerationReverse usingBlock:
            ^(NSString *substring, NSRange substringRange,
              NSRange enclosingRange, BOOL *stop) {
            if ([substring hasPrefix:[document tagMarker]]) {
                [storage deleteCharactersInRange:[storage string]
                    paragraphRangeForRange:substringRange]];
            }
        }];
    #endif
}
return [storage autorelease];

```

This section is separated into two versions, one for the Leopard target and one using a new blocks-based enumeration method for the Snow Leopard target.

The Leopard version uses a traditional technique for removing items from an array or from a string. It calls two new methods, which you will add to the `DiaryDocument` class in a moment, to get the ranges of all of the tag lists in the print view's text storage, saving them temporarily as `NSValue` objects in an array. When that is done, it iterates through the temporary array in reverse order and removes the text in each range. It iterates in reverse order to avoid altering the indexes of the ranges not yet processed.

The Snow Leopard version uses the new `-[NSString enumerateSubstringsInRange:options:usingBlock:]` method, which is simpler and more direct. The method enumerates the string in the print view's text storage by paragraph, as indicated by the `NSStringEnumerationByParagraphs` key, and it does so in reverse order, as indicated by the `NSStringEnumerationReverse` key. It runs the code in the block against each paragraph. The block has two parameters that it uses, `substring` and `substringRange`, and one that it does not use, `stop`. The `substring` is the text of the current paragraph, and the `substringRange` is the range of the current paragraph within the entire text storage. The block directly removes from the print view's text storage every paragraph that begins with the special character returned by `-tagMarker`.

At the end of the `-printableContent` method, you return the fully constructed text storage object. Before too long, you will store this object in an instance variable in the print view for use by other methods.

5. There are a couple of things you need to do to enable the `-printableContents` method to compile and execute. One is to import the `DiaryWindowController` and `DiaryPrintPanelAccessoryController` headers, since the code calls methods

from them. You already imported the `DiaryDocument` header into the `DiaryPrintView`'s header. Add these statements above the `@implementation` directive in the `DiaryPrintView` implementation file:

```
#import "DiaryWindowController.h"
#import "DiaryPrintPanelAccessoryController.h"
```

6. Another thing you must do is to write the two new methods called in the Leopard part of the last section, `+firstTagRangeInString:` and `+nextTagRangeForIndex:inString:`. These are very similar to the `-firstTagRange` and `-nextTagRangeForIndex:` methods that you wrote in Recipe 4. Instead of operating on the diary document's text storage, they take a string parameter and operate on it. They are therefore more generally useful than the original methods, with potentially global application. It is therefore sensible to write them as class methods in the `DiaryDocument` class. Both of them call a common supporting method, `+rangeOfLineFromMarkerIndex:inString:`, which you must also write now.

Because they are essentially identical to the corresponding instance methods you wrote in Recipe 4, they are not given here. Look them up in the downloadable project file for Recipe 9.

You could have avoided writing these class methods had you chosen to implement the Leopard branch of the second block in `-printableContent` using the same algorithm as the Snow Leopard branch—that is, enumerating the paragraphs looking for the special tag marker character at the beginning of each paragraph. However, you already wrote most of the code for the class methods in Recipe 4, and this was the path of least resistance. The only way to know which is faster is performance testing, and you aren't yet ready for that.

7. Now that you have the printable contents, the next task is to paginate them. To take advantage of the highly optimized Cocoa text system, add a layout manager to the print view's text storage, and add a text container object to the layout manager for every page to be printed. Using an `NSTextContainer` object to model each page is a natural application of the Cocoa text system. The text system automatically flows the text in the text storage from text container to text container, just as it does in applications that use text containers to model separate pages or columns of text for onscreen display. The text system contemplates that you would normally create four interacting objects—text storage, layout manager, text container, and text view. However, because you won't need any of the formatting features of an `NSTextView` object here, you don't need to create one.

Before examining the `-paginateContent:` method, remember that the print view's text storage will already have been saved in a `textStorage` instance variable in the print view before this method is called.

In the `DiaryPrintView.h` header file, declare the pagination method immediately following the declaration of `-printableContent`, like this:

```
- (void)paginateContent:(NSSize)defaultPageSize;
```

In the `DiaryPrintView.m` implementation file, define it like this:

```
- (void)paginateContent:(NSSize)defaultPageSize {
    NSLayoutManager *layoutManager = [[NSLayoutManager alloc] init];

    [[self textStorage] addLayoutManager:layoutManager];
    NSTextContainer *textContainer = nil;

    NSUInteger glyphCount = [layoutManager numberOfGlyphs];
    NSUInteger lineLocationInGlyphs = 0;
    NSRange lineRangeInGlyphs;
    NSRect lineRect;
    CGFloat pageHeight;

    do {
        if ((lineLocationInGlyphs == 0) || ((pageHeight +
            lineRect.size.height) > defaultPageSize.height)) {

            if (lineLocationInGlyphs > 0) {
                [textContainer setContainerSize:NSMakeRange([textContainer
                    containerSize].width, pageHeight)];
                [textContainer release];
            }

            textContainer = [[NSTextContainer alloc]
                initWithContainerSize:defaultPageSize];
            [layoutManager addTextContainer:textContainer];
            pageHeight = 0.0;
        }

        if (glyphCount > 0) {
            lineRect = [layoutManager lineFragmentRectForGlyphAtIndex:
                lineLocationInGlyphs effectiveRange:&lineRangeInGlyphs];
            pageHeight += lineRect.size.height;
            lineLocationInGlyphs = NSMaxRange(lineRangeInGlyphs);
        }
    } while (lineLocationInGlyphs < glyphCount);

    [textContainer release];
    [layoutManager release];
}
```

The method receives the default page content size from the caller, which calculated it based on the Print panel's paper size and margin settings.

The initialization section is simple. It creates a new `NSLayoutManager` object and adds it to the print view's existing `NSTextStorage` instance variable using an accessor method you have yet to write. Then it declares all the local variables used in the method. It sets one of them, `glyphCount`, to the total number of glyphs in the printable text storage. This is needed to know when to stop. It sets another, `lineLocation`, to 0. This represents the glyph index of the beginning of each laid-out line of text, which is of course 0 for the first page.

The logic of the `do-while` loop is a little tricky. Follow it closely.

In each iteration, the loop processes a single soft-wrapped line in the printable text storage, starting at the beginning of the text storage and working to the end. It has to compute the dimensions of the current line of text based on the user's Print panel settings for paper width and left and right margins. It also has to break each page vertically at line boundaries based on the paper height and top and bottom margins. In doing all of this, it must also accommodate the user's choice of fonts, font sizes, and styles, which may vary from word to word or even character to character throughout the text storage.

The first part of the loop is responsible for creating new `NSTextContainer` objects and closing out finished `NSTextContainer` objects. The second part of the loop calculates the dimensions of each line, accumulating line heights so that the first part can tell when it needs to close out a finished page and start a new page.

The first part of the loop is executed in the first iteration and in every subsequent iteration wherein the last line processed in the previous iteration would run over the default page boundary. In the first iteration, it skips the code that closes out the previous page, because there is no previous page. It then creates a new page at the default page size and initializes the page height variable to 0.0. After a few more iterations have added enough lines to overrun the default page break, the first part of the loop is executed again. This time, it closes out the just-finished page by adjusting its height to correspond to the bottom of the last line that fit entirely on the page vertically, before creating the next page and reinitializing the page height variable to 0.0. It also releases the memory used by the previous page's local variable. The previous page was retained by the layout manager when it was added to the layout manager, but you didn't release the local variable immediately in order to continue using it.

The second part of the loop isn't executed at all if there are no glyphs because the user is printing a blank page. Otherwise, it is executed for every line. It calculates the `NSRect` required to print the current line, using `NSLayoutManager's -lineFragmentRectForGlyphAtIndex:effectiveRange:` method. It uses the

height of this `NSRect` to increase the `pageHeight` variable. It also uses the `effectiveRange` parameter's indirect return value to calculate the location of the beginning of the next line, placing it in the `lineLocationInGlyphs` variable. Taking the `NSMaxRange()` of the line range yields the location of the beginning of the next line, in glyphs.

The `-lineFragmentRectForGlyphAtIndex:effectiveRange:` method generated the glyphs and laid them out for the entire document, line by line. This is necessary in order to soft-wrap every line of text to the width of the page content area, taking into account font sizes, styling, and anything else that affects the dimensions of a line. For efficiency, the Cocoa text system remembers the layout, and glyph-related methods that you call later will not cause the document content to be laid out again.

When all of the glyphs in the text storage have been processed, the method releases the last text container and the layout manager.

8. Now that the printable text is in hand and paginated according to the Print panel settings, it can be printed. Interestingly, for every time it gets printed onto paper, it may first get printed several times into the print preview. If your application's Print panel includes a preview, the preview is refreshed interactively when the Print panel is opened and again every time the user pages through the print preview or changes a Print panel setting that affects layout.

For a multipage document like the diary document, the Cocoa printing system calls several methods in the print view, and it calls them in a fixed order. Unless you are relying on Cocoa's automatic pagination, you must override three of them, `-knowsPageRange:`, `-rectForPage:`, and `-drawRect:`. The others are optional. Here are all of the methods in the order in which they are called: `-knowsPageRange:`, `-beginDocument`, `-rectForPage:`, `-beginPageInRect:atPlacement:`, `-drawPageBorderWithSize:`, `-drawRect:`, `-endPage`, and `-endDocument`.

The remaining tasks in this step focus on the methods needed to print the document's content. After printing of document content is working, you will arrange to print headers and footers in Step 6. Finally, you will master the difficult topic of print scaling in Step 7.

Start by writing your override of `-knowsPageRange:`. It is called at the beginning of the printing session. It does two things to get the process started. First, it returns as its direct result a `BOOL` indicating whether your application knows enough to be able to handle pagination itself. If it doesn't, you should either write this method to return `NO` or don't override it at all. The printing system will then handle pagination automatically. Here, it returns `YES`, and it therefore also returns a value indirectly in the range parameter, telling the printing system

how many pages the fully paginated printable content will print. The printing system uses this range value to determine how many times to call all of the page-related methods listed earlier.

Your implementation of `-knowsPageRange:` is quite easy to write now, because you've already constructed the printable content and broken it into individual text containers representing pages. All you have to do is call the `-printableContent` and `-paginateContent:` methods you just wrote and then count the text containers. In applications that have fixed page sizes, it is even easier. They can just divide the fixed page height into the total height of the printable content after laying it out.

Insert this method after the Accessor Methods section in the `DiaryPrintView.m` implementation file:

```
#pragma mark OVERRIDE METHODS
```

```
- (BOOL)knowsPageRange:(NSRangePointer)range {
    [self setCurrentOperation:[NSPrintOperation currentOperation]];
    NSPrintInfo *printInfo = [[self currentOperation] printInfo];
    NSSize paperSize = [printInfo paperSize];
    NSSize pageSize = NSMakeSize(paperSize.width - [printInfo leftMargin]
                                - [printInfo rightMargin], paperSize.height - [printInfo
                                topMargin] - [printInfo bottomMargin]);

    [self setTextStorage:[self printableContent]];

    [self setFrame:NSMakeRect(0.0, 0.0, pageSize.width, 1.0e7)];

    [self paginateContent:pageSize];
    NSUInteger pageCount = [[[[[self textStorage] layoutManagers]
                             objectAtIndex:0] textContainers] count];

    range->location = 1;
    range->length = pageCount;

    return YES;
}
```

This method starts by getting the current print operation, which is available globally through an `NSPrintOperation` class method, `+currentOperation`. It immediately sets the `DiaryPrintView`'s `currentOperation` instance variable, using the setter method you wrote at the beginning of this step, so that you don't have to call the global class method in each of the subsequent printing methods you are about to write.

It then gets the print info object from the current print operation. You will do this at the beginning of several of the methods you are about to write, because you often need to refer to the Print panel's current settings. Here, you use the print info to get the current paper size and margin settings from the Print panel, and you do the arithmetic to calculate the size of the page content area.

In the next block, you set the print view's text storage by calling the `-printableContent` method that you just wrote. You haven't yet written the accessor method to set the text storage instance variable, but you will do that in a moment.

You then set the print view's frame. Remember that you set its frame to `NSZeroRect` in `DiaryDocument`. You did this in order to defer to the print view the decision regarding the actual frame size. If you leave the frame set to `NSZeroRect` now, the Cocoa printing system will not print anything. It also won't print anything if you set it to an absurdly large value like `CGFLOAT_MAX`.

It turns out that the print view's frame does not have to be set to any particular value. The pagination and printing scheme you're writing does not use the frame information for any purpose. You therefore simply set its width to the constant width of the page content area. You set its total height to an arbitrarily large value that in practice will not impose any limitations, namely, `1.0e7`, which is suggested in Apple's recent `TextSizing Example` sample code. Doing this enables the printing system to call the remaining methods you are about to write, including the `-drawRect:` method that actually draws the printable content of every page.

Next, you call the `-paginateContent:` method you just wrote, and you set the page count by counting the text containers. You return that value indirectly in the length member of the method's `range` parameter, and you return `YES` directly from the method.

Somewhere in this method or earlier in the printing process you might call `NSView`'s `-setHorizontalPagination:` and `-setVerticalPagination:` methods. You don't call them here because they are set by default to the values you want: `NSClipPagination` horizontally and `NSAutoPagination` vertically. These settings have the effect of printing a single column of vertically paginated pages. You don't need horizontal pagination for the Chef's Diary because, as in most text documents, the text wraps to the page content width.

9. You already added the `textStorage` instance variable and its accessor methods at the beginning of this step. You call the setter once in the `-knowsPageRange:` method, and you use it throughout the print operation. You could not initialize it in the `-initWithFrame:` method because, as you know from the `-printOperationWithSettings:error:` method you wrote in `DiaryDocument`, you

instantiate and initialize the print view before you run the print operation. Therefore, at the time `-initWithFrame:` is called, the print view does not yet have access to the print operation's `printInfo` dictionary, and the print view therefore can't construct the printable contents yet. It first gains access to this information in the `-knowsPageRange:` method.

10. Next in line in the print session is the `-beginDocument` method, which the printing system calls after `-knowsPageRange:`. Overriding this method is optional. It's a good place to set values affecting the printout that don't have an impact on page size. By placing such code in `-beginDocument`, you get it out of `-knowsPageSize:`, allowing the latter method to focus solely on settings that do affect page size and, thus, the page count. If you do override `-beginDocument`, you must call the superclass's implementation because it sets up the current graphics context for printing.

Here, you set printing parameters that turn off centering. In Vermont Recipes, you are exercising total control over the positioning of content on the printed page. By turning off centering, you avoid seeing the last, partial page of your printout centered vertically.

Add this method to the `DiaryPrintView.m` implementation file, after the `-knowsPageRange:` method:

```
- (void)beginDocument {
    [super beginDocument];

    NSPrintInfo *printInfo = [self currentOperation] printInfo];
    [printInfo setHorizontallyCentered:NO];
    [printInfo setVerticallyCentered:NO];
}
```

11. Next up in the print session is `-rectForPage:`, which the printing system calls after `-beginDocument`, once for every page. You must override this method if you are handling pagination yourself. It returns the `NSRect` of the current page in the print view's coordinates, which you will use in `-drawRect:` shortly to draw the page. The `y` member of the origin of the current page should be based on the cumulative height of all previous pages, or `0.0` for the first page. You must include any code here that affects the size of the current page. You can also include other page-specific code here, but you can defer it to `-beginPageInRect:atPlacement:` just as you deferred nonlayout code regarding the document as a whole from `-knowsPageRange:` to `-beginDocument`.

The page argument is one-based, so be careful to subtract 1 before using it with arrays that take a zero-based index.

Insert this method in the `DiaryPrintView.m` implementation file, after the `-beginDocument` method:

```
- (CGRect)rectForPage:(NSInteger)page {
    NSLayoutManager *layoutManager =
        [[[self textStorage] layoutManagers] objectAtIndex:0];
    NSArray *textContainerArray = [layoutManager textContainers];
    NSTextContainer *thisTextContainer;
    CGFloat pageOriginY = 0.0;
    for (NSUInteger idx = 0; idx < page; idx++) {
        thisTextContainer = [textContainerArray objectAtIndex:idx];
        if (idx < page - 1)
            pageOriginY += [thisTextContainer containerSize].height;
    }
    return CGRectMake(0.0, pageOriginY,
        [thisTextContainer containerSize].width,
        [thisTextContainer containerSize].height);
}
```

In an application that has fixed-height pages, this method typically calculates the vertical member of the current page's origin by multiplying the fixed page height by the number of pages minus one. The Chef's Diary document pages are of varying height in order to place page breaks at line boundaries, so instead you iterate over the layout manager's array of text containers, accumulating the actual height of each. You skip the last page.

The math probably puzzles you. That's because an important aspect of the print view hasn't been mentioned: The view is flipped. Normally, Cocoa views use Cartesian coordinates, where the origin is at the bottom-left corner. However, some of the Cocoa text system methods you use require the view to be flipped, with the origin at the top-left corner. When placing marks on a printed page, it is conceptually easier to calculate placement from the top down. The origin for printing does not include the height of the last page because the origin is at the top. Similarly, the origin of the first page is at `{0.0, 0.0}`, and its height is irrelevant in a flipped coordinate system.

To cause the print view to be flipped, you must override `NSView`'s `-isFlipped` method and return `YES`. Add it at the top of the Override Methods section of the `DiaryPrintView.m` implementation file, like this:

```
- (BOOL)isFlipped {
    return YES;
}
```

12. The next method called by the Cocoa printing system is `-beginPageInRect:atPlacement:`. Don't override it now because you don't yet have any code that needs to be executed just before each page is drawn. You will override it in Step 7 when you implement print scaling.
13. You aren't yet ready to override the next method in the print session, `-drawPageBorderWithSize:`, either. You will implement page headers and footers in Step 6.
14. The print session next invokes `-drawRect:`, which actually draws the content of each page in turn. Since you have already written the methods that do all the hard work, this one is very simple. It was provided as a stub method by the template. Flesh it out in the `DiaryPrintView.m` implementation file, after the `-rectForPage:` method, like this:

```
- (void)drawRect:(NSRect)dirtyRect {
    NSLayoutManager *layoutManager =
        [[[self textStorage] layoutManagers] objectAtIndex:0];
    NSUInteger currentPage = [[self currentOperation] currentPage];
    NSRange glyphRange = [layoutManager glyphRangeForTextContainer:
        [[layoutManager textContainers] objectAtIndex:currentPage - 1]];
    NSRect pageRect = [self rectForPage:currentPage];
    [layoutManager drawGlyphsForGlyphRange:glyphRange
        atPoint:NSMakePoint(pageRect.origin.x, pageRect.origin.y)];
}
```

All this does is obtain the glyph range for the current printing page within the printable content, and then draw the glyphs at the origin of the current page within the coordinate system of the view. You get the origin of the current page by calling the `-rectForPage:` method you just wrote. `NSLayoutManager`'s `-glyphRangeForTextContainer:` generates glyphs and lays them out only if they have not already been laid out. Since they were laid out in `-paginateContent:`, no extra layout occurs here.

Ignore the `dirtyRect` argument. It is intended to be used for speeding up drawing to the screen, and you don't need it for printing.

15. You don't need to do any cleanup in either of the final two methods called during a printing session, `-endPage` and `-endDocument`, so don't override them.

That's it for printing the document content. In Steps 6 and 7, you will print page headers and footers and implement print scaling.

Step 6: Print Custom Headers and Footers

While the Cocoa printing system offers you many methods to override in order to print page content, it provides only three override methods for headers and footers: `-drawPageBorderWithSize:`, `-pageHeader`, and `-pageFooter`.

As its name indicates, the `-drawPageBorderWithSize:` method is not restricted to printing headers and footers. You can also print any kind of mark anywhere in the page. The name of the method is actually a little misleading. Although it allows you to print in what are normally thought of as the margins bordering the page content, it also allows you to overprint the page content itself in order to add watermarks or similar features.

Think of the *border* as an enclosing rectangle that encompasses the full sheet of paper. This includes the areas around the edges that are not imageable on most printers unless you elect to print borderless. It also includes the margin areas of the page and the page content area. In fact, the `borderSize` argument to the method is identical to the print info's paper size. It is your responsibility to determine what to print and where to print it within the entire sheet of paper.

Headers and footers are a special case. To print headers and footers at all, you must first set the value for the print info's `NSPrintHeaderAndFooter` key to YES. By default, if you don't override them, the `-pageHeader` and `-pageFooter` methods then return default text, and the Cocoa printing system automatically prints them at default positions in the margin areas at the top and bottom of the page. To control the content of the header and footer, you must override `-pageHeader`, `-pageFooter`, or both. To control their positions, you must draw them yourself in an override of `-drawPageBorderWithSize:`. You can even vary the content and position of the headers and footers from page to page.

The header and the footer are, by default, broken into three components: a left-aligned header or footer, a centered header or footer, and a right-aligned header or footer. The default header and footer contain two tab stops that position and align the centered and the right-aligned header and footer. The position of the left-aligned header and footer is determined by the point at which the header or footer is drawn. The defaults offer a left header displaying the title of the print job (usually the name of the document or, if it hasn't yet been saved, the name of its window), a right header displaying the date of printing, and a right footer displaying the current page number and the total number of pages. The header and footer are both positioned by default so that the left header starts a considerable distance to the left of the left margin setting and the right header and footer are right-aligned a considerable distance

to the right of the right margin setting. There is also a rather large gap between the header and footer and the page content area. If you find the defaults satisfactory, all you have to do to print headers and footers is turn on `NSPrintHeaderAndFooter`.

In this step, you customize both the content and the positions of the header and footer, and you even change the arrangement of tab stops.

1. To begin, turn on headers and footers. You could do this just about anywhere, even in `DiaryDocument`. Here, you do it at almost the last possible moment, in your override of the `-beginDocument` method in `DiaryPrintView`. By placing the code in this method, you can control whether headers and footers are printed separately for every print session, based on the user's current settings in the Print panel accessory view.

Add the following statements to the end of the `-beginDocument` method in the `DiaryPrintView.m` implementation file:

```
NSNumber *doPrintHeadersAndFooters = [[printInfo dictionary]
    objectForKey:VRDefaultDiaryPrintHeadersAndFootersKey];
[[printInfo dictionary] setObject:doPrintHeadersAndFooters
    forKey:NSPrintHeaderAndFooter];
```

This is broken into two statements to make the logic clear. If you had used the printing system's built-in `NSPrintHeaderAndFooter` key to hold your custom accessory view's setting in the first place, it would require no statements at all because you will read its value in your override of the `-drawBorderWithSize:` method. Since you instead declared a custom key, you retrieve its current value here from the `printInfo` object you already set up in `-beginDocument`, and then you transfer it to the built-in key. The printing system is now primed to print headers and footers.

2. Before implementing the `-drawBorderWithSize:` method, override the header and footer methods. Your custom methods add a left tab stop to position the left header and footer, so when you get to `-drawBorderWithSize:`, you will have to draw them in a different position from where you would draw the default header and footer.

Start with the footer because it's a little simpler. It uses the default footer as its basis and adds a left tab stop. Add this method to the `DiaryPrintView.m` implementation file before the existing `-drawRect:` method:

```
- (NSAttributedString *)pageFooter {
    NSMutableAttributedString *footer = [[super pageFooter] mutableCopy];

    NSMutableParagraphStyle *paragraphStyle =
        [[NSParagraphStyle defaultParagraphStyle] mutableCopy];
    [paragraphStyle setTabStops:[self headerAndFooterTabStops]];
```



```

[footer setAttributes:[NSDictionary dictionaryWithObjectsAndKeys:
    paragraphStyle, NSParagraphStyleAttributeName,
    [NSFont systemFontOfSize:0], NSFontAttributeName, nil]
    range:NSMakeRange(0, [footer length])];
[paragraphStyle release];

return [footer autorelease];
}

```

You create a mutable copy of the default footer, which is not mutable itself, so that you can make changes to it. You then create a mutable copy of the Cocoa text system's default `NSParagraphStyle` object for the same reason.

You replace the default paragraph style's tabs by creating your own tab array and installing it using `NSParagraphStyle`'s `-setTabStops:` method. You will write the `-headerAndFooterTabStops` method in a moment to create the custom tab array.

Finally, you install the new tab stops and also replace the default font by calling `NSParagraphStyle`'s `-setAttributes:` method. You obtain the system font using `NSFont`'s `+systemFontOfSize:` method, passing `0` as the size to signal that you want the default size of the font. The system font is Lucida Grande at 13 points. You then release the copy you made of the paragraph style, and you autorelease your modified copy of the footer as you return it.

Notice that you returned an `NSMutableAttributedString` footer even though the return type of the `-pageFooter` method is `NSAttributedString`. Since `NSAttributedString` is the superclass of `NSMutableAttributedString`, this is correct. You could make an immutable copy of the footer first, then release the mutable copy, and then return the immutable copy autoreleased, but this is a little extra work for no practical gain. Callers are expected to honor the declared return type of the method, treating the returned value as an immutable attributed string.

3. Now write the `-headerAndFooterTabStops` method. Near the end of the `DiaryPrintView.h` header file, at the beginning of the Utility Methods section, declare it:

```
- (NSArray *)headerAndFooterTabStops;
```

Define it in the `DiaryPrintView.m` implementation file:

```

- (NSArray *)headerAndFooterTabStops {
    NSPrintInfo *printInfo = [[self currentOperation] printInfo];
    NSTextTab *leftTab =
        [[NSTextTab alloc] initWithType:NSLeftTabStopType
        location:[printInfo leftMargin]];

```

(code continues on next page)

```

NSTextTab *centerTab =
    [[NSTextTab alloc] initWithType:NSCenterTabStopType
     location:[printInfo paperSize].width / 2];
NSTextTab *rightTab =
    [[NSTextTab alloc] initWithType:NSRightTabStopType
     location:[printInfo paperSize].width - [printInfo rightMargin]];
return [NSArray arrayWithObjects:leftTab, centerTab, rightTab, nil];
}

```

The text system maintains a paragraph style object that contains an array of tab stops of any number and kind you desire. Cocoa's `NSTextTab` lets you set each tab stop's alignment using constants like `NSLeftTabStopType`, as well as its location.

Here, you define the left tab stop's type to be left aligned and its location to be the current left margin setting. In your override of `-drawBorderWithSize:`, you will always draw the header and footer up against the left edge of the sheet of paper. Adding a left tab stop and positioning it at the left margin setting therefore aligns the left header and the left footer at the left margin, just as the printing system's default header and footer position the left header and the left footer at their default locations.

From the developer's perspective, adding a left tab stop to the header and footer thus moves the job of positioning all three elements of the header and footer horizontally into the `-pageHeader` and `-pageFooter` methods, allowing you to ignore horizontal positioning issues in the `-drawBorderWithSize:` method and requiring you to deal only with vertical positioning issues there. The printing system's default header and footer, containing only center and right tab stops, require you to think about horizontal positioning in all three methods. The price you pay for this change is a little greater complexity when it comes to scaling, as you will see in Step 7.

4. Turn to the header, where you make more substantial changes than you made in the footer. The `-pageHeader` method is quite long, so look it up in the downloadable project file for Recipe 9.

The `-pageHeader` method begins by setting `date` and `dateLabel` local variables to either the current date as the date the document is being printed, or the date the document was last modified as the date saved, depending on the current setting of the `VRDefaultDiaryPrintTimestampKey` in the Print panel's custom accessory view. The code is straightforward.

The statements setting the `displayDate` local variable use a new Snow Leopard convenience method when the application is running under Snow Leopard. You wrote similar code in Recipe 4.

Next, instead of making a copy of the default header and modifying it, as you did in your override of `-pageFooter`, you create and initialize a new string consisting of three tab characters and the text of the left header and the right header. Using `NSString`'s workhorse `+stringWithFormat:` method, you provide this format string: `@\"\\t%@\\t\\t%\": %@\"`. Each `\\t` sequence is an escaped tab character, one to take advantage of each of the three tab stops you install in the paragraph style in the following statements. Each `%@` sequence is, as you know, a placeholder for an object, in this case a string object. The `“: ”` sequence, a colon followed by a space character, prints as literal text. The text items to be plugged into the placeholders are, for the left header, the print job title, just as in the printing system's default header, and for the right header, a date consisting either of the date printed or the date saved.

Finally, you create the same paragraph style you created for the footer, install it, and return the autoreleased header.

5. Now you can override the `-drawPageBorderWithSize:` method. There is very little to it. It simply calculates the position of the header's and footer's origin on the sheet of paper, leaving a defined gap between the header and footer and the page content, and draws them. This is all about vertically positioning the header and footer. Since you already arranged to position them horizontally using tab stops, you draw both of them up against the left edge of the sheet in this method.

There is one important wrinkle. When calculating the vertical position of the header and footer, you must treat the coordinate system's origin as being at the bottom-left corner of the sheet even though the view is flipped. The `-drawPageBorderWithSize:` method handles headers and footers both for text documents and for nontext documents, so it can't assume that the view is flipped. The Cocoa default is to use nonflipped views.

In the `DiaryPrintView.m` implementation file, insert this method before the `-pageHeader` and `-pageFooter` methods:

```
- (void)drawPageBorderWithSize:(NSSize)borderSize {
    NSPrintInfo *printInfo = [[self currentOperation] printInfo];
    if ([[printInfo dictionary] objectForKey:NSPrintHeaderAndFooter]
        boolValue) {
        NSRect imageablePageBounds = [printInfo imageablePageBounds];
        CGFloat spacingFactor = 1.5;
        CGFloat originX = 0.0;
```

(code continues on next page)

```

CGFloat headerOriginY = borderSize.height - [printInfo topMargin]
    + ([[self pageHeader] size].height * spacingFactor);
CGFloat imageablePageTop = imageablePageBounds.origin.y +
    imageablePageBounds.size.height;
if (headerOriginY + [[self pageHeader] size].height
    > imageablePageTop) {
    headerOriginY =
        imageablePageTop - [[self pageHeader] size].height;
}
[[self pageHeader] drawAtPoint:
    NSMakePoint(originX, headerOriginY)];

CGFloat footerOriginY = [printInfo bottomMargin]
    - ([[self pageHeader] size].height * spacingFactor)
    - [[self pageHeader] size].height;
CGFloat imageablePageBottom = imageablePageBounds.origin.y;
if (footerOriginY < imageablePageBottom) {
    footerOriginY = imageablePageBottom;
}
[[self pageFooter] drawAtPoint:
    NSMakePoint(originX, footerOriginY)];
}
}

```

The method first checks the print info to make sure the custom accessory view in the Print panel is set to request headers and footers in the printout, using the `NSPrintHeaderAndFooter` key.

If so, it first sets three local variables that will be used without change in subsequent calculations. The first sets the `imageablePageBounds` rectangle. You need it to make sure the header and footer are not positioned outside the printable area of the page for the selected printer. The next, `spacingFactor`, isolates the value for the gap between the header and footer and the page content in a separate setting that is easy to change if you decide you don't like it. Here, you call for a gap of one and one-half line heights. Third, you set the x element of the origin for both the header and the footer to `0.0`, since horizontal placement of header and footer elements is controlled by tab stops.

The next block of statements sets the header's vertical origin and draws it. Counting from the sheet's origin at the bottom, you get the paper height in order to move to the top of the sheet, subtract the top margin to get to the location of the top of the page content, and then add one and one-half line heights.

This is the vertical origin for the header, and you could stop here if it weren't for the area around the sheet on which the printer can't print. You add some statements that limit the vertical origin of the header in such a way that the top of the header will always print within the imageable page bounds.

The final block of code performs a similar calculation for the footer, and then draws it.

6. Printing in the page border is not limited to headers and footers. As noted above, you can print anything you like in the margins, and you can even overprint the page content in the middle of the page. For an example, you will now print a square block of four characters in each of the four corners of the page. All you have to do is calculate the origins of each corner mark in such a way that it fits on the page. For this exercise, you disregard the imageable bounds. The corner marks appear only in the print preview in the Print panel, in PDF files, and on paper if you print borderless.

Insert this code at the beginning of the `-drawPageBorderWithSize:` method:

```
NSString *cornerMark = @"XX\nXX";
[cornerMark drawAtPoint:NSZeroPoint withAttributes:nil];
[cornerMark drawAtPoint:NSMakePoint(0.0, borderSize.height
    - [cornerMark sizeWithAttributes:nil].height)
    withAttributes:nil];
[cornerMark drawAtPoint:NSMakePoint(borderSize.width
    - [cornerMark sizeWithAttributes:nil].width, 0.0)
    withAttributes:nil];
[cornerMark drawAtPoint:NSMakePoint(borderSize.width
    - [cornerMark sizeWithAttributes:nil].width,
    borderSize.height - [cornerMark sizeWithAttributes:nil].height)
    withAttributes:nil];
```

Printing a watermark over the page content is left as an exercise for the reader.

Step 7: Implement Print Scaling

In my experience, there are two reasons for scaling a printout down and two reasons for scaling a printout up. When scaling down, either you want to fit more content on a sheet of paper, or you want to fit the existing page content onto a smaller sheet. In either case, you're presumably stuck with the size of the paper, and in the latter case you're prepared to fold or cut it down to the smaller size. When scaling up, either you want to fit less

content on a sheet of paper but keep the amount of white space around the edges the same, or you want to fit the existing page content onto a larger sheet, such as a banner. Again, you're presumably stuck with the size of the paper, and in the latter case you're printing on a continuous roll or you're prepared to tape multiple sheets together.

The subject matter and intended use of a document should have some bearing on how it is scaled for printing. It makes good sense to fit more of a spreadsheet's content onto a sheet, for example, because the user can then scan a row across all of its columns without having to turn the page and line it up with the next page. But it usually makes no sense at all to fit more text across a sheet, because it's very hard to scan a long line of prose.

Examining print scaling behavior in a range of popular applications suggests that developers often don't think this through as well as they could. Consider scaling down in these examples, all from Apple:

- TextEdit anchors the scaled page content to the top-left corner of the page content area. It leaves the margins unscaled. While it scales the text of the headers and footers, it leaves them anchored to the four corners of the sheet. Depending on certain preference settings and the settings in a couple of menus, it sometimes fills the page content area from margin to margin both horizontally and vertically, bringing in additional content from the next page—but it sometimes does not. I'm not clear on what settings have what effect in this regard.
- Preview, when printing PDF files, centers the scaled page content both horizontally and vertically. It does not offer headers and footers, which is reasonable since they are often provided by the PDF file itself.
- Safari anchors the scaled page content to the top-left corner of the page content area, like TextEdit. However, it scales the margins too, unlike TextEdit. While it scales the text of the headers and footers, it always leaves them anchored to the four corners of the sheet, like TextEdit. Like TextEdit on some settings, it fills the sheet with more content if the Web page is big enough.
- Mail, when printing an e-mail message, centers the scaled page content horizontally but not vertically. Vertically, it anchors the text to the top margin. It leaves the margins unscaled. It does not offer headers and footers.

When scaling up, there are other differences between these applications.

- TextEdit expands the number of pages so that you can print the entire document. It leaves the margins unscaled.
- Preview does not expand the number of pages, so you lose content. It makes the margins smaller.

- Safari expands the number of pages so that you can print the entire document. It makes the margins smaller.
- Mail expands the number of pages so that you can print the entire e-mail message. It makes the margins smaller.

It is difficult to rationalize all of these behaviors. For example, why would a user ever want to scale the page content down but leave the headers and footers anchored to the four corners of the sheet? Then folding or cutting the sheet of paper to the size of the scaled page content loses the headers and footers. Why would a user ever want the margins scaled? Then, depending on the printer, some of the page content around the edges won't print if the edges get too close to the edge of the sheet. Why would a user ever want the scaled-down page content centered in either dimension? Then all four edges have to be cut or folded, instead of only two of them. Why would a user ever want text to be added at the bottom from the next page? Then the sheet can't be folded or cut down proportionally without losing content. Why would a user ever want text scaled down in size but rewrapped to the full original width of the page content area? It is well understood that people have difficulty reading overly long lines of text. And why would a user ever want scaled-up text to be lost because the number of pages was not increased?

In *Vermont Recipes*, you assume that the user's only interest in scaling down the text of the *Chef's Diary* is to fold or cut the sheets into a smaller size—for example, to fit them into a small index card box. You therefore solve the scaling-down problem as follows: by anchoring scaled text to the upper-left corner of the page content area so that only two sides need to be folded or cut; by moving the footer up and the right header left so that they still appear on the smaller sheet after it is folded or cut; by preserving the margin sizes so that none of the page content becomes unprintable; and by leaving every page's content on its own page. You solve the scaling-up problem by not allowing it. Why would anybody ever want to print the *Chef's Diary* on a banner?

Print scaling requires you to add code to several of the methods that you have written, so it is a little more difficult than adding headers and footers to your application's printing repertoire. The difficulty is compounded by the fact that the header and footer coordinate system is not flipped but the page content in a text-based application is flipped, so you have to approach scaling from two different angles. Finally, getting the math right can be mind-boggling anyway, because some values, like the print margins in print info, are not scaled, while other values, like the dimensions of your page content area, are scaled. Sometimes you have to divide by the scaling factor, and sometimes you have to multiply by the scaling factor. Sometimes you scale content, and sometimes you scale the space between or around content. You have to think it through systematically. A trial-and-error approach based on intuition is not workable.

1. The actual scaling of the page content is done in an override of `-beginPageInRect:atPlacement:`. Add this method to the `DiaryPrintView.m` implementation file after the `-rectForPage:` method:

```
- (void)beginPageInRect:(NSRect)rect atPlacement:(NSPoint)location {
    [super beginPageInRect:rect atPlacement:location];

    NSPrintInfo *printInfo = [[self currentOperation] printInfo];
    CGFloat scalingFactor;
    if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
        scalingFactor = [[[printInfo dictionary]
                          objectForKey:NSPrintScalingFactor] floatValue];
    } else {
        scalingFactor = [printInfo scalingFactor];
    }
    NSAffineTransform *transform = [NSAffineTransform transform];
    [transform scaleBy:scalingFactor];
    [transform concat];
}
```

As with `-beginDocument`, you must always call the superclass's implementation, because it sets up the coordinate system and performs translation of the view.

With that out of the way, you access the print info object to get the current scaling factor as set in the Print panel. In Snow Leopard, the scaling factor is available through a new method on `NSPrintInfo`, `-scalingFactor`. In Leopard, you must obtain it from the print info dictionary using the `NSPrintScalingFactor` key. You use either method here, depending on which version of the operating system is running.

Finally, create an `NSAffineTransform` object, and call its `-scaleBy:` method to scale the view in both dimensions proportionally according to the scaling factor. Apply the transform using the `-concat` method. To scale a view, you don't need to know anything more about transforms than this.

You still have a lot of work to do, though, because at this stage the application puts the scaled page content, the headers and footers, and the corner marks in the wrong places. For one thing, the Cocoa printing system automatically scales the border content—the headers and footers and the corner marks—without making you use a transform, but it does so in a coordinate system that is anchored to the bottom-left corner of the sheet of paper. If you scale the print job to 50%, for example, the border content is moved down to the lower-left quadrant of the sheet, while the page content is moved up to the upper-left quadrant because the view is flipped.

Furthermore, the headers and footers are moved closer to the edges of the sheet, possibly encroaching on the nonimageable area, while the page content's margins are not scaled. The reason for this discrepancy is that the border content has no margins; it encompasses the entire sheet, so everything on it is scaled.

You have to add code to several methods to put everything in its proper place.

2. Start by dealing with scaling up to more than 100%. You simply disallow this, for the reasons discussed at the beginning of this step. I'm not aware of any way to control the user's entry of a scaling value in the Scale text field in the Print panel, so if a user types, say, 150%, the field will accept and display that value. The user deserves an explanation as to why the print preview does not immediately scale up, so you should provide an alert explaining the situation.

Add the following statements near the beginning of the `-knowsPageRange:` method in the `DiaryPrintView.m` implementation file, after the statement defining the `printInfo` local variable:

```
CGFloat scalingFactor;
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
    scalingFactor = [[[printInfo dictionary]
        objectForKey:NSPrintScalingFactor] floatValue];
    if (scalingFactor > 1.0) {
        [[printInfo dictionary] setObject:[NSNumber numberWithInt:1.0]
            forKey:NSPrintScalingFactor];
    }
} else {
    scalingFactor = [printInfo scalingFactor];
    if (scalingFactor > 1.0) {
        [printInfo setScalingFactor:1.0];
    }
}

if (scalingFactor > 1.0) {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    if (![defaults boolForKey:
        DEFAULT_ALERT_PRINT_SCALED_UP_SUPPRESSED_KEY]) {
        NSAlert *alert = [self alertCannotPrintScaledUp];
        [alert runModal];
    }
}
```

(code continues on next page)

```

        if ([[alert suppressionButton] state] == NSOnState) {
            [defaults setBool:YES forKey:
                DEFAULT_ALERT_PRINT_SCALED_UP_SUPPRESSED_KEY];
            [defaults synchronize];
        }
    }
}

```

Define the key by adding this at the top of the `DiaryPrintView.m` implementation file, before the `@implementation` directive:

```
#pragma mark MACROS
```

```
#define DEFAULT_ALERT_PRINT_SCALED_UP_SUPPRESSED_KEY
    @"alert print scaled up suppressed"

```

The first section is broken into two parts only to show the different ways of limiting the effective scaling value in Leopard and Snow Leopard. The first part uses the `NSPrintScalingFactor` key, while the second part uses the Snow Leopard `-setScalingFactor` method. Both parts reset the internal scaling value to **1.0** if the user enters a value greater than 100%.

The second section presents an application-modal alert explaining that scaling above 100% is not allowed. It presents the alert only if the user has not previously elected to suppress it. You have already used this technique once, in Recipe 7, to allow the user to suppress the alert informing the user that the diary document was restored after a crash. In Recipe 7, you used an alert sheet; here, you have to use a modal alert because the Print panel is already open as a sheet. Apple strongly discourages presenting a sheet on another sheet, and I judge that the user experience would be too awkward if you closed the Print panel just to show this alert as another sheet.

Now code the modal alert itself. Declare its method at the end of the `DiaryPrintView.h` header file like this:

```
#pragma mark ALERTS
```

```
- (NSAlert *)alertCannotPrintScaledUp;
```

Define it in the `DiaryPrintView.m` implementation file like this:

```
#pragma mark ALERTS
```

```
- (NSAlert *)alertCannotPrintScaledUp {
    NSBeep();
}

```

```

    UIAlertView *alert = [[[UIAlertView alloc] init] autorelease];
    [alert setMessageText:
        NSLocalizedString(@"Cannot print larger than 100%.",
        @"message text for CannotPrintScaledUp alert")];
    [alert setInformativeText:
        NSLocalizedString(@"Prints at 100% when Scale is larger.",
        @"informative text for CannotPrintScaledUp alert")];
    [alert setShowsSuppressionButton:YES];
    return alert;
}

```

Again, you already did this once before, in Recipe 7.

3. Turn next to the border content, in the `-drawPageBorderWithSize:` method. There are two issues here: the positions of the corner marks and the positions of the headers and footers.

The corner marks were added as an afterthought to demonstrate that this method can print more than headers and footers. Take the easy way out for now and simply suppress printing the corner marks when scaling to less than 100%. If you think the corner marks should be printed to assist the user in folding or cutting the scaled-down printout, you can scale them using the same technique you will apply in a moment to the headers and footers.

To suppress the corner marks when scaling, simply enclose the block of code near the beginning of the `-drawPageBorderWithSize:` method that draws them in an `if` block that allows them to be drawn only when the scaling factor is exactly 100%. The revised block should look like this:

```

CGFloat scalingFactor;
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
    scalingFactor = [[[printInfo dictionary]
        objectForKey:NSPrintScalingFactor] floatValue];
} else {
    scalingFactor = [printInfo scalingFactor];
}
if (scalingFactor == 1.0) {
    NSString *cornerMark = @"XX\nXX";
    [cornerMark drawAtPoint:NSZeroPoint withAttributes:nil];
    [cornerMark drawAtPoint:NSMakePoint(0.0, borderSize.height
        - [cornerMark sizeWithAttributes:nil].height)
        withAttributes:nil];
}

```

(code continues on next page)

```

[cornerMark drawAtPoint:NSMakePoint(borderSize.width
    - [cornerMark sizeWithAttributes:nil].width, 0.0)
    withAttributes:nil];
[cornerMark drawAtPoint:NSMakePoint(borderSize.width
    - [cornerMark sizeWithAttributes:nil].width,
    borderSize.height - [cornerMark sizeWithAttributes:nil].height)
    withAttributes:nil];
}

```

You again use two different techniques to get the current scaling factor, one for Leopard and one for Snow Leopard.

4. Now you get to the hard part, repositioning the headers and footers when the scaling factor is less than 100%. This requires some mental gymnastics, because you don't want to scale some features—the margins—whereas you do want to scale others. Furthermore, different considerations apply to horizontal scaling and vertical scaling in the case of the headers and footers.

Tackle the horizontal dimension first. Recall that your custom header and the footer contain a left tab stop that is positioned at the left margin, and as a result, the header and footer as a whole must be printed up against the left edge of the paper, with a horizontal origin of `0.0`. That works only at 100% scaling. At, say, 50% scaling, the displayed text of the header and footer is scaled to half its normal size—including the effective width of the tab stops contained within the header and footer. Therefore, if you were to print the header and footer at horizontal location `0.0`, the left tab stop would be repositioned to a point somewhere to the left of the left margin. Since you want the left margin to remain unscaled, you have to adjust the drawing origin by moving it to the right from the left edge of the sheet. To do this, you must reset the left origin of the header and footer by subtracting the unscaled left margin from the value obtained by scaling the left tab stop's width back up to 100%. Here's the code snippet that accomplishes this:

```

CGFloat originX = ([printInfo leftMargin] / scalingFactor)
    - [printInfo leftMargin];

```

Note that, when the scaling factor is 100%, this results in `0.0`, as it must for the header and footer to be positioned correctly at 100% scaling. At less than 100% scaling, it yields a positive number representing the rightward adjustment of the origin required to make the scaled left tab stop come out at the unscaled margin.

Now tackle the vertical dimension. The math is different for the header and for the footer. Start with the header. Counting from the unflipped vertical origin at the bottom of the sheet of paper, measure up to the top of the sheet unscaled,

measure down (that is, subtract) the height of the unscaled top margin, and then divide the result by the scaling factor in order to scale the numbers up. The paper height and the top margin values in print info were not changed when you scaled the border area—that is, when you scaled the entire sheet of paper—so you have to increase the values in order to come out where you want to end up in the scaled printout, namely, at the unscaled top of the page content area, which is the top margin. Next, measure up (that is, add) the line height of the header times the spacing factor, to get the gap between the top margin and the vertical origin of the header. You have to adjust the gap to make it shorter vertically on account of the scaling, which means in this case that you must multiply the spacing factor by the scaling factor instead of dividing by the scaling factor. Here’s the code snippet that does this:

```
CGFloat headerOriginY = ((borderSize.height - [printInfo topMargin])
    / scalingFactor) + ([[self pageHeader] size].height
    * spacingFactor * scalingFactor);
```

Again, note that when the scaling factor is 100%, this comes out as the top margin plus the line-and-a-half gap. When the scaling factor is less than 100%, it comes out a little lower, or closer to the top margin, than that, which is exactly what you specified.

Before printing the document, you must check to make sure the header does not run over the imageable boundary at the top of the sheet. Since scaling has the effect of moving the header downward, closer to the unscaled top margin, you don’t have to adjust this test for the scaling factor. If the header passes the test when it is unscaled, it is guaranteed to pass the test when it is scaled below 100%. However, the actual value of the imageable page bounds must be scaled upward, because the size of the paper in the printer doesn’t change when you scale the view. To scale it upward, you divide by the scaling factor. Here is the snippet that accomplishes this:

```
CGFloat imageablePageTop = (imageablePageBounds.origin.y
    + imageablePageBounds.size.height) / scalingFactor;
if (headerOriginY + ([[self pageHeader] size].height / scalingFactor)
    > imageablePageTop) {
    headerOriginY = imageablePageTop - ([[self pageHeader] size].height
    / scalingFactor);
```

If the obviousness of these three code snippets doesn’t pop right out at you, think about it. Seriously, you have to master the conceptual complexity in order to avoid wasting a lot of time working out the formulas. To test yourself, figure out why the vertical adjustment to the footer origin works, in the final version reproduced next.

Here is the entire section of the `-drawPageBorderWithSize:` method that handles the positioning of the header and footer. Insert it at the end of the method in the `DiaryPrintView.m` implementation file, after the code dealing with the corner marks:

```
if ([[printInfo dictionary] objectForKey:NSPrintHeaderAndFooter]
    boolValue)) {
    NSRect imageablePageBounds = [printInfo imageablePageBounds];
    CGFloat spacingFactor = 1.5;
    CGFloat originX = ([printInfo leftMargin] / scalingFactor)
        - [printInfo leftMargin];

    CGFloat headerOriginY = ((borderSize.height - [printInfo topMargin])
        / scalingFactor) + ([[self pageHeader] size].height
        * spacingFactor * scalingFactor);
    CGFloat imageablePageTop = (imageablePageBounds.origin.y
        + imageablePageBounds.size.height) / scalingFactor;
    if (headerOriginY + ([[self pageHeader] size].height / scalingFactor)
        > imageablePageTop) {
        headerOriginY = imageablePageTop
            - ([[self pageHeader] size].height / scalingFactor);
    }
    [[self pageHeader] drawAtPoint:NSMakePoint(originX, headerOriginY)];

    CGFloat footerOriginY = ((borderSize.height - [printInfo topMargin])
        / scalingFactor) - borderSize.height + [printInfo topMargin]
        + [printInfo bottomMargin] - ([[self pageHeader] size].height
        * spacingFactor) - ([[self pageHeader] size].height
        * scalingFactor);
    CGFloat imageablePageBottom = imageablePageBounds.origin.y;
    if (footerOriginY < imageablePageBottom) {
        footerOriginY = imageablePageBottom;
    }
    [[self pageFooter] drawAtPoint:NSMakePoint(originX, footerOriginY)];
}
```

5. After that, scaling and positioning the page content in the `-drawRect:` method will seem easy. All you have to do is get the current scaling factor from print info, and then divide it into the height of the `dirtyRect` argument. Recall that

the page content is flipped, with its origin at the top-left corner of the sheet of paper. The vertical origin is not 0.0 except on the first page of a multipage document. On subsequent pages, the vertical origin is the height of the top page break of the current page in the overall height of the entire document, measured from the top. This means that you are dividing the unscaled page break value by the scaling factor in order to scale up the vertical dimension of the page content.

Modify the `-drawRect:` method to this, in the `DiaryPrintView.m` implementation file:

```
- (void)drawRect:(NSRect)dirtyRect {
    NSLayoutManager *layoutManager =
        [[[self textStorage] layoutManagers] objectAtIndex:0];
    NSUInteger currentPage =
        [[NSPrintOperation currentOperation] currentPage];
    NSRange glyphRange = [layoutManager glyphRangeForTextContainer:
        [[layoutManager textContainers] objectAtIndex:currentPage - 1]];
    NSRect pageRect = [self rectForPage:currentPage];

    NSPrintInfo *printInfo = [[self currentOperation] printInfo];
    CGFloat scalingFactor;
    if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
        scalingFactor = [[[printInfo dictionary]
            objectForKey:NSPrintScalingFactor] floatValue];
    } else {
        scalingFactor = [printInfo scalingFactor];
    }

    [layoutManager drawGlyphsForGlyphRange:glyphRange
        atPoint:NSMakePoint(pageRect.origin.x,
            pageRect.origin.y / scalingFactor)];
}
```

You are now finished with Recipe 9 except for testing.

Step 8: Build and Run the Application

You haven't done much intermediate testing in this recipe because the code is very interdependent. So give the Print panel a good workout.

First, build and run both application targets, the Vermont Recipes SL target and the Vermont Recipes target. You should again test them on three different platforms: a PowerPC Mac running Leopard, an Intel Mac running Leopard, and an Intel Mac running Snow Leopard.

Start your testing by creating a new Chef's Diary document and filling it with several entries, at least some of them with tag lists, and a lot of text in at least some of the entries. You want to end up with a document that will fill at last two sheets of paper or, preferably, more, and which has entries filling the page content area horizontally from margin to margin. Select some text and leave it selected. Save the document so that you can easily reopen it for more testing without having to add the content again.

Then choose File > Print (**Figure 9.7**). Unless you have already reset them on your own, all of the settings in the panel are at their default values, both in the top area and in the custom accessory view. The print preview on the left reflects these values. The first page is shown, and its page content area is full, stretching from the top margin to the bottom margin and from the left margin to the right margin. No tag lists are shown. The headers and footers appear near the top and bottom: left and right headers and a centered footer. The four corner marks appear in the corners.

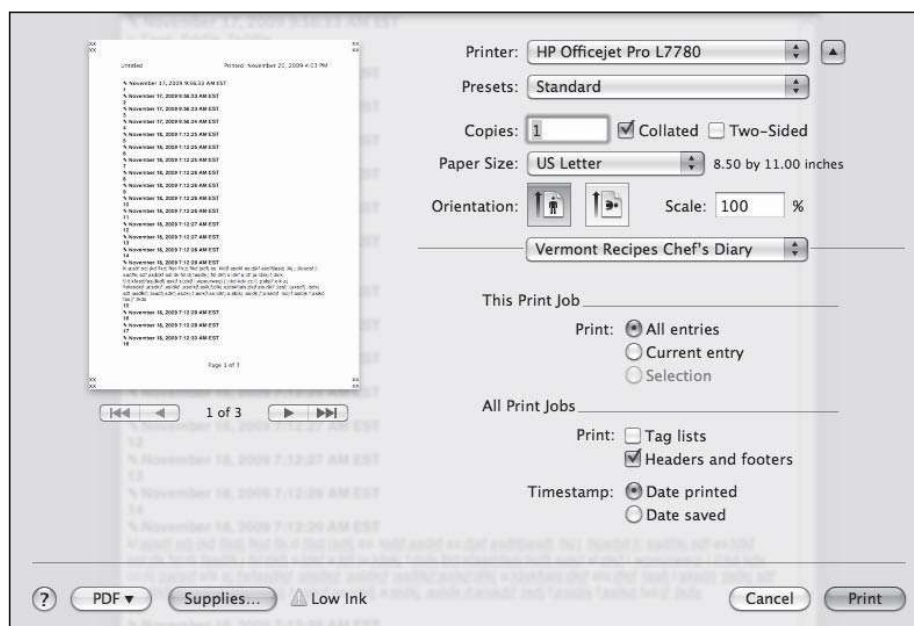


FIGURE 9.7
The Print panel
with default
settings.

Click a control beneath the print preview to show the last page. It only partially fills the page content area, and it starts at the top margin (**Figure 9.8**).

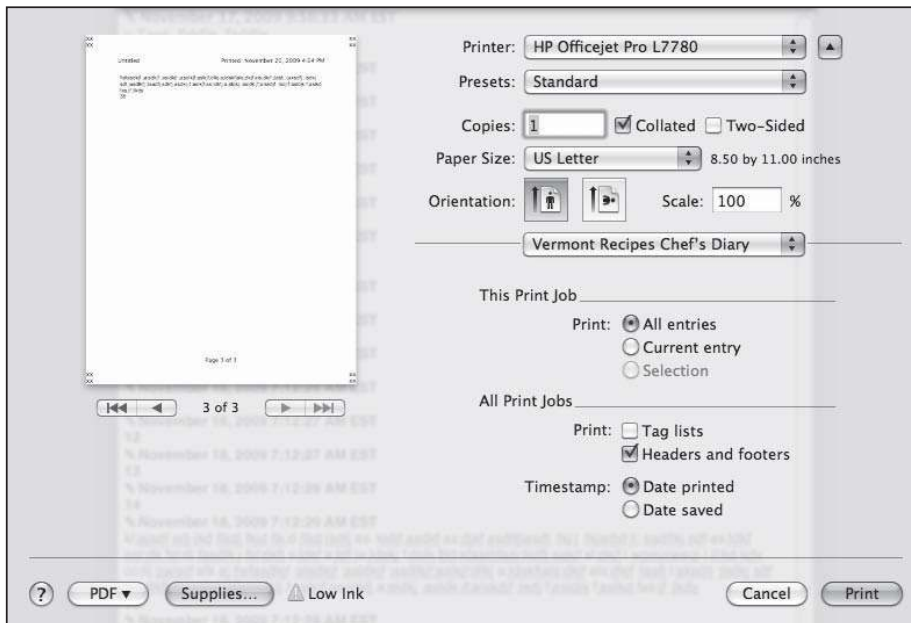


FIGURE 9.8
The Print panel showing the last page of the document.

Go to a page that has a lot of text on it, and click controls in the standard settings area in the top-right area of the Print panel. For example, click the landscape orientation button, and the print preview flips sideways. All of the page's constituents are correctly positioned. The body text of entries with text extends across the entire widened page content area, as does the header. The corner marks are still in the corners of the sheet (**Figure 9.9**).

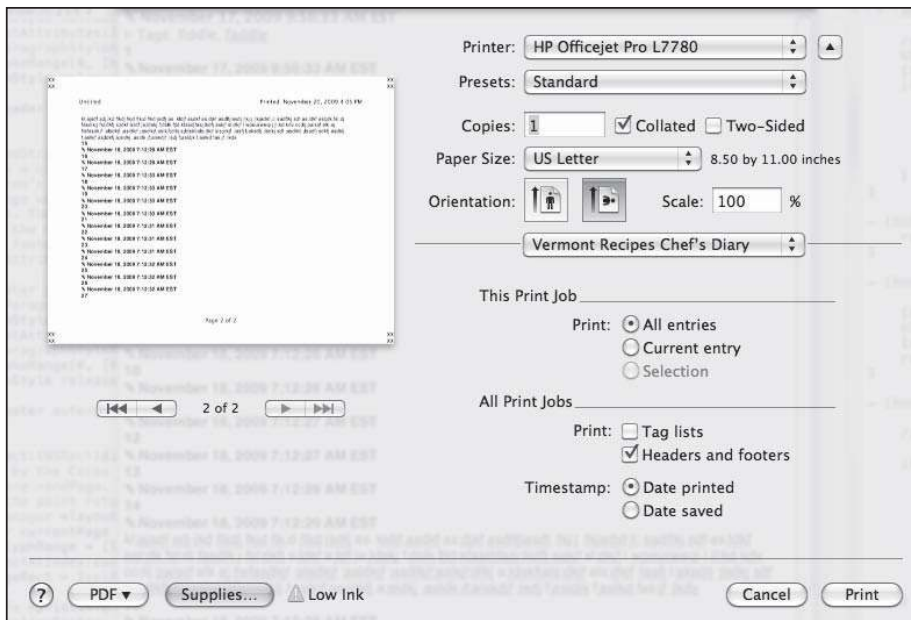


FIGURE 9.9
The Print panel set to landscape orientation.

Enter **75%** in the Scale text field. The entire page, both page content and header and footer, gets smaller and is positioned correctly in the top left of the sheet, respecting the unscaled margins (**Figure 9.10**).

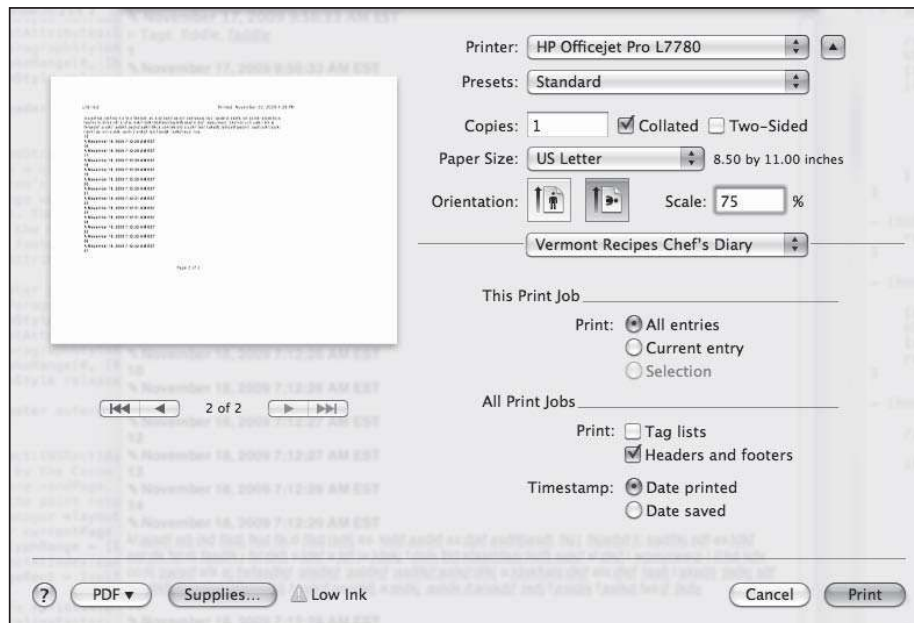


FIGURE 9.10
The Print panel scaled to 75%.

Return to portrait orientation and 100% scaling. Then select the “Current entry” radio button, and the print preview instantly shows only the current entry. Select the Selection radio button, and only the selection shows. Go back to “All entries,” and then select the “Tag lists” checkbox, and all of the tag lists suddenly appear in the print preview. Deselect the “Headers and footers” checkbox, and the headers and footers disappear. Reselect the “Headers and footers” checkbox (**Figure 9.11**). Then click the “Date saved” radio button, and if it isn’t too small, you see the date change from Printed to Saved.

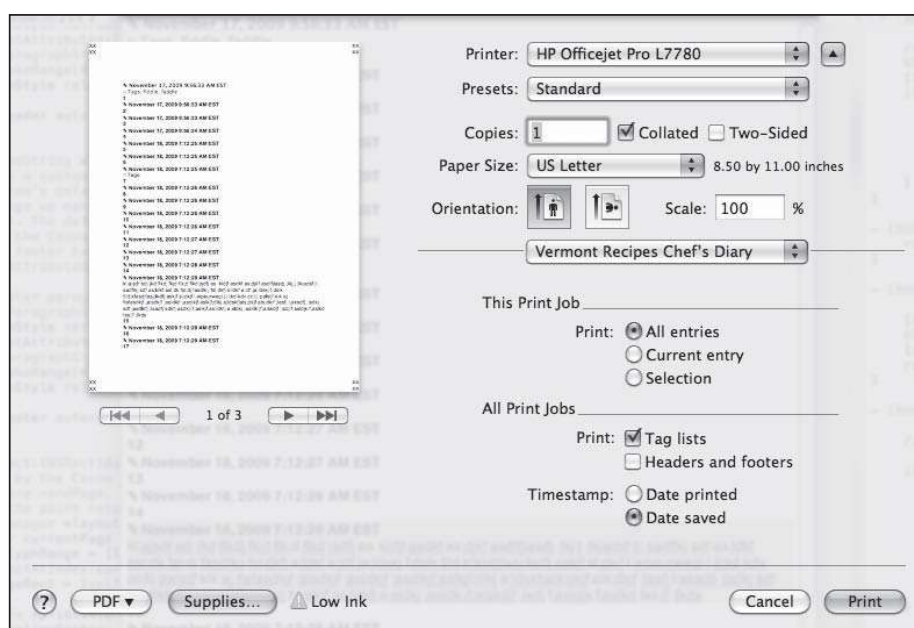


FIGURE 9.11
The Print panel with custom settings.

Close the Print panel and immediately reopen it. All of the settings have returned to their default values, including the “All entries” radio button, except that the settings in the All Print Jobs section of the custom accessory view remain at their changed values, as designed.

One thing you might make a note of for a future version of Vermont Recipes is that none of the settings in the custom accessory view can be saved in a preset. At least the settings in the All Print Jobs section of the panel probably should be eligible for inclusion in a preset. In Leopard and Snow Leopard, you make settings eligible for saving in a preset by placing them not in the print info’s `dictionary` dictionary but in its `printSettings` dictionary. Settings stored in the `printSettings` dictionary are synchronized with their counterparts in the `dictionary` dictionary. The Presets pop-up menu in the Print panel and the `printSettings` dictionary involve Core Printing, formerly known as the Print Manager, which is beyond the scope of this book.

Step 9: Save and Archive the Project

Quit the running application. Close the Xcode project window, discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like **Vermont Recipes 2.0.0–Recipe 9.zip**. The working Vermont Recipes project folder remains in place, ready for Recipe 10.

Conclusion

In this recipe, you added the first of several features that belong in almost any real application: printing support. Several more recipes will similarly focus on a single feature. In Recipe 10, you turn to application preferences.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 9.

Class Reference and Protocol Documents

NSDocument Class Reference

NSViewController Class Reference

NSPrintPanel Class Reference

NSPrintPanelAccessorizing Protocol Reference

NSPrintInfo Class Reference

NSPrintOperation Class Reference

NSLayoutManager Class Reference

NSObject Class Reference

NSParagraphStyle Class Reference

General Documentation

Mac OS X Developer Release Notes: Cocoa Application Framework (10.5 and Earlier)

Printing Programming Topics for Cocoa

Cocoa Fundamentals Guide (Creating a Singleton Instance)

Mac OS X SnowLeopard Release Notes: Cocoa Application Framework

Cocoa Drawing Guide (Coordinate Systems and Transforms)

Text System Overview

Text System User Interface Layer Programming Guide for Cocoa (Setting Text Margins)

Text Layout Programming Guide for Cocoa

Text System Storage Layer Overview

Core Printing Reference

Technical Note TN2248: Using Cocoa and Core Printing Together

Technical Note TN2155: Saving Printer Settings for Automatic Printing

Add a Preferences Window

Recipe 10, like Recipe 9, focuses on a single topic. In this case, it's preferences.

Every Macintosh application except the very simplest has use for a preferences window. The need is so common that the Xcode template for document-based applications includes by default a Preferences menu item, in the MainMenu nib file's application menu, to open a preferences window. The Preferences menu item comes with the standard Command-comma keyboard shortcut. It is not connected to a default action method to open the preferences window because there are many different ways you might choose to implement a preferences system in your application.

Your job is to create the preferences window and its associated preferences window controller, and to write and connect a simple IBAction to open the preferences window. You must also, of course, add user interface elements for all of the application settings that you want the user to be able to change, and to associate them with entries in the application's user defaults database. You have already had some experience with the Cocoa frameworks' NSUserDefaults class, so the techniques used to set initial default values and to read and write the user's settings will be familiar. The techniques you learn in this recipe will allow you to implement preferences of any degree of complexity in your applications.

HIGHLIGHTS

Adding support for application preferences with the user defaults database

Implementing a preferences window

Using a tab view and tab view items

Changing a window's title when its content changes

Resetting an alert's suppression button in preferences

Synchronizing the preferences window with an alert's suppression button

Using the -windowDidBecomeKey: delegate method

Using the NSUserDefaultsDidChangeNotification notification

Changing the standard state of a window in preferences

Configuring a number formatter in Interface Builder

Setting printing preferences

Synchronizing the preferences window with a Print panel

Changing document autosaving intervals in preferences

Consider what features you want the user to be able to set as application preferences. There are two features that you already decided to implement. In Recipe 6, you decided to allow the user to choose any existing Chef's Diary file (maybe even any PDF file) as the current Chef's Diary. In Recipe 7, you decided to allow the user to change the diary document's autosave delay interval, or even to turn off autosave. There are additional possibilities. For example, the Print panel accessory view you created in Recipe 9 allows the user to set the content when printing the diary document. Three of those settings persist across multiple print jobs, so they are in effect user preferences. It might be convenient to allow the user to set them in the application's print window, too. In addition, you created global variables in Recipe 7 to determine the standard size of the Chef's Diary window and the recipes window, and you could allow the user to reset the standard sizes of those windows in the preferences window. Finally, you implemented two warning alerts, in Recipes 7 and 9, that the user is allowed to suppress, and it might be useful to allow the user to turn them back on.

Step 1: Design and Build a Preferences Window in Interface Builder

The preferences window is a standard window, and it is modeless. An application's preferences window should not be modal because the user may need to use other application windows alongside it in order to understand what settings to enter. Alerts and dialogs can be document-modal sheets or application-modal freestanding panels, but in both cases the user's access to some features of the application is blocked while the panel is open. You create and use a modeless preferences window just as you'd create and use any standard window.

1. Leave the archived Recipe 9 project folder in the zip file where it is, and open the working Vermont Recipes subfolder.

Increment the version in the Properties pane of the Vermont Recipes target's information window from 9 to **10** so that the application's version is displayed in the About window as *2.0.0 (10)*.

2. Use Interface Builder to design and build the user interface of the preferences window. Like most windows, the preferences window should have its own nib file, which you can create from the Xcode File menu. This helps to encapsulate the design of the preferences window, and it makes for more efficient memory use because the application won't have to load the preferences window if the user doesn't open it.

In Xcode, choose File > New. In the New File dialog, select User Interface in the source list, select Window XIB in the upper-right pane, and click Next. In the next dialog, name the file **PreferencesWindow.xib**. Make sure that both the Vermont Recipes and Vermont Recipes SL targets are selected, and then click Finish. If necessary, move the new PreferencesWindow.xib file to the Resources group in the Groups & Files pane of the project window, below DiaryPrintPanelAccessoryView.xib.

3. Double-click the PreferencesWindow.xib file in Xcode to open it in Interface Builder. Then choose Window > Document Info. In the PreferencesWindow.xib Info window, set the Deployment Target to Mac OS X 10.5 and set the Development Target to Default – Interface Builder 3.2.
4. The “Preferences Windows” section of the *Apple Human Interface Guidelines (HIG)* describes a preferences window as a modeless dialog. It must not be resizable, and its zoom and minimize buttons should be disabled. If it contains a toolbar, the toolbar should not be customizable. The window’s title in the title bar should be the name of the currently selected pane or, if there is only one pane, the name of the application followed by “Preferences.” When the preferences window is closed and reopened, it should reopen to the same pane that was selected when the user closed it, at least while the application remains running. The menu command to open the preferences window must be named Preferences, it must be in the application menu, and it must have a Command-comma keyboard shortcut. When the user makes changes to settings in the preferences window, the changes should take effect immediately, without requiring the user to click an OK or Apply button and without waiting for the user to close the window.

You can implement some of these requirements in Interface Builder. In the Window Attributes inspector, deselect the Resize and Minimize checkboxes in the Controls section and the Shows Toolbar Button checkbox in the Appearance section.

5. The *HIG* notes that many applications separate the contents of their preferences windows into separate panes, each representing a functional category selectable by clicking a button in a toolbar. The *HIG* does not mandate use of a toolbar, however. According to the “Tab Views” section of the *HIG*, a tab view or a segmented control is acceptable. They allow the same separation into separate panes as a toolbar, but without requiring you to hire an artist to design toolbar items. Furthermore, a tab view or a segmented control looks better than a half-empty toolbar when there are only two or three separate panes. Even if there are many panes, the *HIG* reluctantly allows you to avoid a toolbar and instead to use a pop-up menu. You use a tab view in Vermont Recipes.

In Interface Builder, drag a Tab View object from the Layout Views subsection of the Cocoa Views & Cells section of the Library and drop it in the design surface.

Then adjust all four sides so that the edges (including the top edge of the tabs) coincide with the guides that appear as you drag. The *HIG* allows you to extend the left, bottom, and right edges of a tab view to the edges of the design surface, if you prefer, but you must then leave a margin of at least 20 pixels between the user interface elements in the tab view and the edges of the design surface.

The tab view comes with two tabs, but you need three tabs. Select the tab view, and in the Tab View Attributes inspector, change the Tabs field from 2 to 3. Double-click each tab in turn to edit the titles, and name them *General*, *Recipes*, and *Chef's Diary*, from left to right.

Select the tab view, and in the Tab View Connections inspector, drag from the little circle to the right of the **delegate** outlet to the File's Owner proxy in the nib file's document window. You will use an NSTabView delegate method in the next step.

6. For the moment, the only user interface elements required in the General pane are two checkboxes to turn on or off the alerts that appear when the user attempts to scale a printed document above 100% or when the application restores an autosaved document. In Recipe 7, you alerted the user when an autosaved Chef's Diary document was restored. In Recipe 9, you disallowed scaling above 100% when the user prints the Chef's Diary document, and you alerted the user when an attempt to do that is made. You included the suppression checkbox in both alerts. You will do the same in the similar alerts that you have yet to write for the recipes document. The same principles apply to both documents, so it makes sense to turn all four of these alerts on or off using checkboxes in the application-wide General preferences pane.

Some of the panes in the preferences window will have multiple controls relating to differing topics, so take this into account when adding these checkboxes by making it a discrete section of the pane. Start by providing a title for this section. Select the General tab view, and drag a label object from the Inputs & Values subsection of the Library and drop it in the tab view's content view, positioning it near the left and top edges based on the guides. You can verify that you dropped it in the correct view by clicking it while holding down Shift and Control. You should see a list showing, from top to bottom, the window, its content view, the top tab view, another view, and the new label field. Double-click the label to select it for editing, and enter **Alerts**. Select the text, and press Command-B to make it bold.

Next, drag a checkbox object from the Inputs & Values subsection of the Library and drop it in the tab view's content view, positioning it below the Alerts label based on the guides. In the Button Attributes inspector, deselect the State checkbox in the Visual section to indicate that, by default, the alert will not be suppressed. Double-click the checkbox to select its text for editing, and name it **Don't show alert when attempting to print larger than 100%**. Option-drag the

checkbox downward to position a copy and rename it **Don't show alert when restoring from autosaved document** (Figure 10.1).

Leave the section label left aligned in its tab pane, but center the two checkboxes.



FIGURE 10.1 The General pane of the preferences window.

7. You haven't done much with the recipes document yet, but in Recipe 7 you did set its standard size. Make the standard size a settable preference for the recipes document now.

Select the Recipes tab. Add a label near the top and name it **Recipes Window** using the same technique you used to add the Alerts label to the General pane. Drag another label below it and name it **Standard size:**. Drag two text fields and two stepper controls, and lay them out to the right of the Standard size label, arranged on the model of the width and height fields in the Window Size inspector. Drag two more labels and center them immediately below the two text fields, naming them **Width** and **Height**, respectively. Finally, drag a push button, name it **Use Current Size**, and position it below the Width field.

You should install a number formatter in the width and height fields to limit the user's options to reasonable values. Drag number formatter objects from the Formatters subsection of the Library, and drop one in each field. Configure them almost identically. In the Behavior section at the top of the Number Formatter Attributes inspector, choose Mac OS X 10.4+ Custom. Select the Grouping Separator checkbox so that separators (commas in the United States) appear if the user sets the width or height to a really large number. Deselect the Allows Floats checkbox so that the user cannot enter fractional pixels. In the Constraints scrolling view at the bottom of the inspector, set the Maximum to **10000** because the window server limits window sizes to 10,000 pixels in each dimension.

The one difference in formatter settings as between the two fields is the Minimum constraint. Set the Minimum to **700** for the Width field and **350** for the Height field. These are the minimum dimensions that you set for the recipes window in Recipe 7. It's easier to enforce this constraint in the nib file than in

code. When the user attempts to enter a value that is less than the minimum, the application will beep when the user attempts to commit the value.

Select the Width and Height fields in turn, and in the Text Field Attributes inspector, select the right-aligned button.

Select each of the steppers in turn, and in the Stepper Attributes inspector, set the Maximum value to **10000.00** and the Increment to **10.00**. Although a stepper doesn't display a value, it does contain a value, and a stepper increments or decrements its value every time the user clicks the top or bottom arrow. You will use the values of these two steppers in Step 2 to change the values of the corresponding text fields. You will see that it is important to keep the value of each stepper synchronized with its corresponding text field. You set the increment to 10.00 instead of the default 1.00 to speed up the response when the user holds down a stepper arrow for continuous change. Also set the Minimum value in each stepper to the minimum value in the number formatter attached to the associated text field, **700** for width and **350** for height.

Leave the section label left aligned, but center the user interface element group (**Figure 10.2**).



FIGURE 10.2 The Recipes pane of the preferences window.

8. You have done a lot of work with the Chef's Diary document, so there are several preferences settings to add to the Chef's Diary pane.

Start by setting up a Chef's Diary Window section that is identical to the Recipes Window section you just set up for the Recipes pane, except for its section label. You can do this by dragging to select all of the user interface elements you just added to the Recipes pane, choosing the Chef's Diary pane, clicking repeatedly until the view within the top tab view is selected, and then pasting. Reposition the group using the guides, and rename the section label **Chef's Diary Window**. Check the values in the number formatters and in the steppers to be sure they are correct, and change the Minimum width and height in the diary window number formatters to **550** and **550**, respectively, as you set them

in Recipe 7. Also set the Minimum value in each stepper to the minimum value in the number formatter attached to the associated text field, **550** for width and **550** for height.

Next, drag a horizontal line object from the Layout Views subsection of the Library, position it below the Chef's Diary Window section's user interface elements, and extend its ends to the margins. Below that, add a section label and name it **Printing**. The settings for this section should be identical to those in the All Print Jobs section of the diary document's accessory print view. Open the DiaryPrintPanelAccessoryView nib file. Drag to select the two checkbox objects, the radio button object, and the two labels in the All Print Jobs section. Copy them to the clipboard, and then return to the Chef's Diary pane of the PreferencesWindow nib file and paste them below the section title.

Add another horizontal line object; then add a new section label and name it **Autosaving**. Add a label and a pop-up button from the Inputs & Values subsection of the Library. Enter **Autosave documents:** for the label. Double-click the pop-up button to open it, and name the first three menu items **Every 15 seconds**, **Every 30 seconds**, and **Every minute**. Option-drag the third menu item twice to create two more menu items, and name them **Every 5 minutes** and **Never**. Choose "Every 30 seconds" as the default. This pop-up button is identical to that in the New Document pane of TextEdit's preferences window.

Finally, add a section to the Chef's Diary pane to select the current Chef's Diary. If you need to make the preferences window larger, drag its resize control as appropriate; then select the tab view and Command-drag its bottom and right edges as appropriate. Add a horizontal line, and a section label reading **Document**. Add a label named **Current Chef's Diary:** and a text field to its right. You could use a path control here instead of a text field, but I'm old fashioned.

Leave the labels and dividers left aligned, but center the user interface elements within the window in accordance with the *HIG* (**Figure 10.3**).

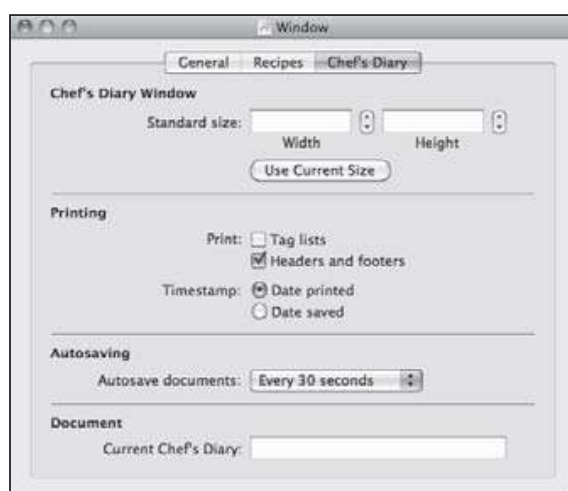


FIGURE 10.3 The Chef's Diary pane of the preferences window.

9. The user interface elements in the preferences window don't require help tags because their wording and labels are clear. However, you should connect the accessibility titles as appropriate. For the width field at the top of the Recipes pane, for example, Control-drag from the text field to the Width label beneath it and select "accessibility title" in the HUD, and then Control-drag from the width label to the "Standard size" label and repeat the process. Repeat the process with each of the other settings elements that has a title.
10. Configure the preferences window's initial first responder and its key view loop. You learned about the initial first responder and the key view loop in Step 1 of Recipe 4. Remember that you should connect all user interface elements, not just text fields, in the key view loop, in case the user sets Full Keyboard Access to "All controls" in the Keyboard Shortcuts tab of the Keyboard pane in System Preferences.

Some conceptual complexity results from the presence of the tab view in the preferences window. Because the tab view is at the top of the window, it should be the window's first responder. You can't connect a tab view item to another tab view item as next key view, because you use the arrow keys instead of the tab key to move from one tab view item to another, and Cocoa handles this automatically. However, you should set an initial first responder for each tab view item. The key view loop should form a complete circle among the user interface items within the tab view item, excluding the tab. After the user selects one of the tabs, tabbing within that tab view item then starts with the initial first responder, cycles through all the other user interface items within the tab view item, and returns to the tab before starting the circle again.

Start by setting the window's initial first responder, which should be the tab view. Select the window in Interface Builder by clicking its title bar; then go to the Window Connections inspector. Drag from the little circle beside the `initialFirstResponder` outlet to any of the tabs in the design surface. The tab view is now the window's first responder. If you pause a moment while holding the pointer over one of the tabs, that tab view item becomes selected, but the tab view itself is still designated as the window's first responder.

Select the General tab view item by clicking it twice. Go to the Tab View Item Connections inspector and drag from the circle beside the `initialFirstResponder` outlet to the top checkbox. Then create a complete circle of `nextKeyView` connections from the top checkbox to the bottom checkbox and from the bottom checkbox back to the top checkbox.

Perform the same tasks in the Recipes and Chef's Diary tab view items, and you're done. You actually could have skipped the initial first responder and the key view loop entirely, because you laid out the preferences window logically with a strict top-to-bottom and left-to-right orientation, and Cocoa got the key view loop right automatically. You should at least remember to test every window's key view loop with "All controls" turned on to make sure you're happy with it.

11. Select the General tab view item before saving the nib file, to make sure that the preferences window opens with it selected when the user first runs the application. Then save the nib file.
12. Select the General tab view, and then save a snapshot. Name it **Recipe 10 Step 1**, and add a comment saying **Created Preferences Window in Interface Builder**.

You will return to the nib file in Step 2 to make connections after you have created the preferences window controller.

Step 2: Create a Preferences Window Controller in Xcode

With the preferences window's user interface out of the way, you must next subclass `NSWindowController` to create a customized window controller for it. Name it *PreferencesWindowController*. In it, you will implement accessor methods to get and set the values associated with the user interface elements in the preferences window. You will also add code to save the settings in the user defaults.

In this step, simply set up the basic features of the preferences window controller that allows you to instantiate it and open its window. The preferences window controller has in common with the custom print accessory view controller the twin facts that it is only needed if the user opens its associated window or view and that, once it is created, there is no pressing need to release it until the user quits the application. Like the accessory view controller, it is a singleton, and it can therefore be created lazily by a class factory method the first time it is needed. Thereafter, calling the factory method simply returns the existing window controller. The action method you will connect to the Preferences menu item in the MainMenu nib file need only call the preferences window controller's `+sharedController` factory method to create or get the singleton shared controller instance, and then call the controller's built-in `-showWindow:` action method.

1. In Xcode, choose File > New File. In the New File dialog, select Cocoa Class in the source list and Objective-C class in the upper-right pane, and choose `NSWindowController` from the "Subclass of" pop-up menu. In the next dialog, name the file **PreferencesWindowController.m**, select the "Also create 'PreferencesWindowController.h'" checkbox, make sure both the Vermont Recipes and Vermont Recipes SL targets are selected, and click Finish. If necessary, place the header and implementation files at the end of the Window Controllers group in the Groups & Files pane of the Xcode project window. Set up the information at the top of both files in the usual fashion.

2. In Interface Builder, select the File's Owner proxy in the PreferencesWindow.xib document window; then go to the Object Identity inspector and choose PreferencesWindowController as the Class.
3. Control-drag from the File's Owner proxy to the window icon, and choose the window outlet in the HUD.
4. Control-drag from the window icon to the File's Owner proxy and choose delegate in the HUD.
5. Write the +sharedController factory method. In the PreferencesWindowController.h header file, declare it like this:

```
#pragma mark FACTORY METHOD
```

```
+ (PreferencesWindowController *)sharedController;
```

In the PreferencesWindowController.m implementation file, define it like this:

```
#pragma mark FACTORY METHOD
```

```
+ (PreferencesWindowController *)sharedController {
    static PreferencesWindowController *sharedController = nil;
    if (sharedController == nil) {
        sharedController = [[self alloc]
                           initWithWindowNibName:@"PreferencesWindow"];
    }
    return sharedController;
}
```

This is essentially identical to the +[DiaryPrintPanelAccessoryController sharedController] method that you wrote in Recipe 9.

6. Now write the action method that opens the preferences window. You will connect the action method to the Preferences menu item, which is the application menu and must be available at all times. It must therefore be written in the VRApplicationController class that you created in Recipe 5.

Start by importing the PreferencesWindowController.h header file into the VRApplicationController.m implementation file by adding this at the end of the existing #import directives:

```
#import "PreferencesWindowController.h"
```

Then, in the VRApplicationController.h header file, declare the action method after the two action methods that are already there, like this:

```
- (IBAction)showPreferences:(id)sender;
```


Define it in the `VRApplicationController.m` implementation file, like this:

```
- (IBAction)showPreferences:(id)sender {
    [[PreferencesWindowController sharedController] showWindow:sender];
}
```

Build the project so that the new action method will appear in Interface Builder.

7. Return to the `MainMenu` nib file. Open the Vermont Recipes application menu and select the Preferences menu item. Go to the Menu Item Connections inspector, and drag from the little circle to the right of the selector item in the Sent Actions section to the First Responder proxy in the nib file's document window and choose the `showPreferences:` action. As you know, the application controller is in the responder chain because it is the delegate of the shared application object. Save the nib file.
8. Back in Xcode, there are some details to take care of. First, in the `PreferencesWindowController.m` implementation file, configure some features of the preferences window when the user first opens it, by overriding the `-windowDidLoad` method, like this:

```
#pragma mark OVERRIDE METHODS
```

```
- (void)windowDidLoad {
    NSWindow *window = [self window];
    [window center];
    [window setExcludedFromWindowsMenu:YES];

    NSTabViewItem *selectedTabViewItem = [(NSTabView *)[window
        initialFirstResponder] selectedTabViewItem];
    [window setTitle:[selectedTabViewItem label]];
}
```

Calling `NSWindow`'s `-center` method centers the preferences window horizontally and positions it a little above the center vertically. According to the *HIG*, this is the correct starting position for auxiliary windows like the preferences window.

The `-setExcludedFromWindowsMenu:` method does just what it says. The *HIG* does not indicate whether an application's preferences window should be listed in the Window menu, saying only that document windows should be included but that panels normally are not included. Cocoa automatically includes the preferences window unless you reverse the default using `-setExcludedFromWindowsMenu:` as `TextEdit` does. However, most of Apple's applications include the preferences window in the Window menu, so you can't be faulted if you do so. I choose to follow `TextEdit`'s example because the title of a multi-pane preferences window

changes depending on which pane is selected, and I find it confusing to see these changing names in the Window menu. The Preferences menu item is always available in the application menu when you need it.

The `TextEdit` sample code calls `-setHidesOnDeactivate:NO`, but this is only necessary if the preferences window is based on `NSPanel`, which defaults to `YES`. This preferences window is based on `NSWindow`, which defaults to `NO`. Furthermore, the Window Attributes inspector includes a checkbox to control this setting, so you don't need to set it in code in any event.

Set the window's title when it first opens. The *HIG* requires the title of a multi-pane preferences window to be identical to the title of the current pane. Normally, you would need to declare and connect an `IBOutlet` to access the tab view in order to get the label of its selected tab view item. Here, however, you know that you designed the preferences window so that the tab view would be its initial first responder. It is therefore easier to call the `-initWithFirstResponder` method of `NSWindow`. This requires casting to `NSTabView*` to suppress the warning that `NSView` might not respond to `-selectedTabViewItem`. Be careful with this `-initWithFirstResponder` shortcut, however. It is not very robust from a code maintenance viewpoint, since a future version of *Vermont Recipes* might make some other user interface element the preferences window's initial first responder. To be safe, note in a comment that the tab view is set as the window's initial first responder in Interface Builder.

9. Finally, you must arrange for the preferences window's title to change every time the user selects a different tab view item. `NSTabView` declares several delegate methods, including `-tabView:willSelectTabViewItem:`, to meet just this sort of need. You already set the preferences window controller to be the tab view's delegate in Step 1, so you can implement the delegate method now. Add it at the end of the `PreferencesWindowController.m` implementation file:

```
#pragma mark DELEGATE METHODS
```

```
- (void)tabView:(NSTabView *)tabView
    willSelectTabViewItem:(NSTabViewItem *)tabViewItem {
    [[self window] setTitle:[tabViewItem label]];
}
```

Now, every time the user selects a different tab view item, the window's title updates automatically at the same time. This happens whether the user clicks another tab with the mouse, uses the arrow keys to select another tab, or selects another tab by running an AppleScript script.

10. You can now build and run the application and open the preferences window. Try it. When you choose *Vermont Recipes > Preferences*, the preferences window

opens. While you're there, test the key view loop to be sure it works to your satisfaction, and make sure the window's title changes whenever and however you select a different tab.

In subsequent steps, you will implement each set of preferences in turn.

11. Save a snapshot. Name it **Recipe 10 Step 2**, and add a comment saying, **Created PreferencesWindowController**.

Step 3: Configure the General Tab View Item

The General tab view item contains two checkboxes controlling global application settings. One of them, when set to YES, suppresses the warning alert that is otherwise displayed any time the user attempts to scale a diary document larger than 100% for printing. The other, when set to YES, suppresses the warning alert that is otherwise displayed when the application restores an autosaved document. You already arranged to save these settings in the application's user defaults in Recipes 7 and 9, but only when the user selects the suppression checkboxes in the two warning alerts where they are used. Once the user selects a suppression checkbox, there is no way to turn the warning alert back on because the alert is not displayed again. You must provide a separate user interface element to turn it back on, and the General pane of the preferences window is an appropriate place to do that. The user need only deselect the applicable checkbox in the General pane, and the alert will begin to appear again when appropriate.

When you originally set up the two suppression checkboxes in their alerts, you defined the keys under which they would be saved to the user defaults by using `#define` preprocessor macros. You defined the `DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY` key in the `DiaryWindowController.m` implementation file in Recipe 7, and you defined the `DEFAULT_ALERT_PRINT_SCALED_UP_SUPPRESSED_KEY` key in the `DiaryPrintView.m` implementation file in Recipe 9. These macros are local to the implementation files in which they are defined, so they cannot be used anywhere else. You have adopted a practice of setting up other keys for use with the user defaults by declaring external `NSString` variables in header files so that they are globally available to any other file that imports those header files. You will have to declare similar external variables for use with the two checkboxes in the General pane, after you set up the required accessor and action methods.

To get and set the values of the two checkboxes in the General pane, you need an `IBOutlet` instance variable and a getter accessor method for each of them, and you need an action method for each to set the corresponding values in the user defaults.

1. Declare instance variables for the checkboxes. In the PreferencesWindowController.h header file, between the curly braces of the `@interface` directive, enter the following declarations:

```
IBOutlet NSButton *suppressScaleUpAlertCheckbox;  
IBOutlet NSButton *suppressRestoreAutosavedDocumentAlertCheckbox;
```

2. Declare getter accessor methods at the end of the PreferencesWindowController.h header file as follows:

```
#pragma mark ACCESSOR METHODS
```

```
- (NSButton *)suppressScaleUpAlertCheckbox;  
- (NSButton *)suppressRestoreAutosavedDocumentAlertCheckbox;
```

Define them after the Factory Method section of the PreferencesWindowController.m implementation file as follows:

```
#pragma mark ACCESSOR METHODS
```

```
- (NSButton *)suppressScaleUpAlertCheckbox {  
    return suppressScaleUpAlertCheckbox;  
}  
  
- (NSButton *)suppressRestoreAutosavedDocumentAlertCheckbox {  
    return suppressRestoreAutosavedDocumentAlertCheckbox;  
}
```

3. Now write action methods for each. The job of each action method is very simple: When the user clicks the checkbox, store its new value in the user defaults using the appropriate key.

Declare the action methods at the end of the PreferencesWindowController.h header file, like this:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)setDefaultSuppressScaleUpAlert:(id)sender;  
- (IBAction)setDefaultSuppressRestoreAutosavedDocumentAlert:(id)sender;
```

Define them after the Accessor Methods section of the PreferencesWindowController.m implementation file, like this:

```
#pragma mark ACTION METHODS
```

```
- (IBAction)setDefaultSuppressScaleUpAlert:(id)sender {  
    [[NSUserDefaults standardUserDefaults] setBool:[sender state]  
        forKey:VRDefaultSuppressScaleUpAlertKey];  
}
```

```
- (IBAction)setDefaultSuppressRestoreAutosavedDocumentAlert:(id)sender
{
    [[NSUserDefaults standardUserDefaults] setBool:[sender state]
        forKey:VRDefaultSuppressRestoreDiaryDocumentAlertKey];
}
```

Both action methods use a shortcut you have seen before. They read the current state of the sender—the checkbox that the user just clicked—and treat its new value as a BOOL. Its value is really either `NSOnState` or `NSOffState`, but these values are effectively cast to `YES` or `NO` when passed in the parameter to the `-setBool:forKey:` method. This is safe here because the checkbox was not set up as a mixed-state checkbox.

4. To make the action methods work, you must declare and define the new global keys you used in them. You do this in the files where you previously set them up in Recipes 7 and 9.

Start with the `VRDefaultSuppressScaleUpAlertKey` key. At the top of the `DiaryPrintView.h` header file, before the `@interface` directive, declare it like this:

```
extern NSString *VRDefaultSuppressScaleUpAlertKey;
```

At the bottom of the `DiaryPrintView.m` implementation file, after the `@end` directive, define it like this:

```
NSString *VRDefaultSuppressScaleUpAlertKey =
    DEFAULT_ALERT_PRINT_SCALED_UP_SUPPRESSED_KEY;
```

Do the same with the `VRDefaultSuppressRestoreDiaryDocumentAlertKey` key. At the top of the `DiaryWindowController.h` header file, before the `@interface` directive, declare it like this:

```
extern NSString *VRDefaultSuppressRestoreDiaryDocumentAlertKey;
```

At the bottom of the `DiaryWindowController.m` implementation file, after the `@end` directive, define it like this:

```
NSString *VRDefaultSuppressRestoreDiaryDocumentAlertKey =
    DEFAULT_ALERT_RESTORE_DIARY_DOCUMENT_SUPPRESSED_KEY;
```

In both definitions, you simply reused the macros from Recipes 7 and 9 to set the values of the new global variables.

Later, when you implement the same functionality for the recipes window, you may have to consider renaming this key.

5. To use these global variables in the preferences window controller's action methods, you must import the header files in which you declared them. Do this at the top of the `PreferencesWindowController.m` implementation file:

```
#import "DiaryPrintView.h"
#import "DiaryWindowController.h"
```

6. The action methods still won't work, however, unless you connect them in Interface Builder. Build the application first to make sure the action methods are available in Interface Builder. Then Control-drag from each checkbox in the design surface to the First Responder proxy in the nib file's document window, and then select its action method.
7. A critical step remains—namely, to ensure that the two checkboxes in the General pane reflect the current values in the user defaults database every time the user opens the preferences window. It's easy to do this: Simply set the state of the checkboxes to the values currently existing in the user defaults database.

But where should you place the code that does this? Your first instinct may be to place it in the `-windowDidLoad` method. However, this method is called only when the window's nib file loads, and that typically happens only the first time the user opens the window. When the user subsequently closes the window, the nib file remains loaded and it is not loaded again when the user reopens the window. Thus, the `-windowDidLoad` method isn't called when the user reopens the window. If the user changed the user defaults in the meantime in one of the warning alerts, the reopened preferences window will not reflect the change.

This is a general issue, and it can trip you up in many similar situations. In general, remember that the `-awakeFromNib` and `-windowDidLoad` methods execute only when the nib file is loaded, and this does not necessarily happen every time the nib file's window is closed and reopened, just as the `-init` method is not called every time the window is closed and reopened. As a result, these methods should normally be used only to set the initial state of the window to values that the user cannot change or, if changed, do not need to be reset after the user closes and reopens the window.

You typically deal with this issue the same way you handled renaming the window when the user selected another tab view item: by implementing an appropriate delegate method. For this problem, that would be the `-windowDidBecomeKey:` delegate method. It is called every time the user opens the preferences window, whether or not the nib file is reloaded. It is perfectly tailored for this situation, and as a bonus, it is also called when the preferences window is already open but is brought to the front. You made the preferences window controller the window's delegate in Step 1. Simply add the following method at the end of the Delegate Methods section in the `PreferencesWindow.m` implementation file:

```
- (void)windowDidBecomeKey:(NSNotification *)notification {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [[self suppressScaleUpAlertCheckbox] setState:[defaults boolForKey:
        VRDefaultSuppressScaleUpAlertKey]];
```



```

[[self suppressRestoreAutosavedDocumentAlertCheckbox]
    setState:[defaults boolForKey:
        VRDefaultSuppressRestoreDiaryDocumentAlertKey]];
}

```

This is the task that led you to set up IBOutlets for the two checkboxes in the first place. You didn't need outlets for the action methods, because the action methods' `sender` arguments gave you access to the checkboxes and their current settings. The `-windowDidBecomeKey:` delegate method doesn't know anything about the checkboxes, however, so you need the outlets.

8. The outlets won't work until you connect them, so do it now. In the Preferences-Window nib file, Control-drag from the File's Owner proxy to each checkbox in turn and connect its outlet.
9. The two checkboxes in the General pane will now work as expected if you build and run the application—except in one relatively rare situation. If the preferences window is open and visible at the same time as one of the warning alerts, the checkbox in the preferences window does not change its state when the user changes the state of the corresponding suppression checkbox in the alert and dismisses the alert.

Fixing this requires that the preferences window be notified of any changes made to the user defaults database as a result of external user actions. This is not an unusual problem, and to help you solve it, the `NSUserDefaults` class posts the `NSUserDefaultsDidChangeNotification` notification every time any user defaults setting is changed. If there is a possibility that an external actor can change a user defaults setting, you can update the open preferences window automatically in real time by observing this notification.

Register for the notification at the end of the `-windowDidLoad` method, using the same techniques you used in Recipes 7 and 8 to register for several notifications relating to the diary document. When you use Snow Leopard's blocks-based technique to register for notifications, this is a little more complicated than it is in Leopard, but it is also more robust. For Snow Leopard, you will have to write a little more code.

Start by adding the following registration code at the end of the `-windowDidLoad` method in the `PreferencesWindowController.m` implementation file:

```

NSNotificationCenter *defaultCenter =
    [NSNotificationCenter defaultCenter];
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_6
    [defaultCenter addObserver:self
        selector:@selector(userDefaultsDidChange:)

```

(code continues on next page)

```

        name:NSUserDefaultsDidChangeNotification object:nil];
#else
    [self setUserDefaultsDidChangeObserver:[defaultCenter
        addObserverForName:NSUserDefaultsDidChangeNotification
        object:nil queue:nil
        usingBlock:^(NSNotification *notification) {
            [self userDefaultsDidChange:notification];
        }]];
#endif

```

You did not have to declare and define the `NSUserDefaultsDidChangeNotification` notification name, because it is supplied by Cocoa.

You put the code to be executed into a separate method, `-userDefaultsDidChange:`, since you need to execute the same code whether running Leopard or Snow Leopard. If you were not supporting Leopard and used only the blocks-based registration method, you could move the code from the separate method directly into the block.

Write the `-userDefaultsDidChange:` method now. Declare it at the end of the `PreferencesWindowController.h` header file, like this:

```

#pragma mark NOTIFICATION METHODS
- (void)userDefaultsDidChange:(NSNotification *)notification;

```

Define it at the end of the `PreferencesWindowController.m` implementation file, like this:

```

#pragma mark NOTIFICATION METHODS

- (void)userDefaultsDidChange:(NSNotification *)notification {
    NSUserDefaults *defaults = [notification object];
    [[self suppressScaleUpAlertCheckbox]
        setState:[defaults boolForKey:
            VRDefaultSuppressScaleUpAlertKey]];
    [[self suppressRestoreAutosavedDocumentAlertCheckbox]
        setState:[defaults boolForKey:
            VRDefaultSuppressRestoreDiaryDocumentAlertKey]];
}

```

The notification method is called every time it observes the `NSUserDefaultsDidChangeNotification` notification, which is posted every time any setting in the user defaults is changed. This means that the method may be called when unrelated settings are changed, but this does no harm because the two statements in the method are very fast and they simply set the state of the checkbox to its cur-

rent state. The real work happens when the notification is posted because one of the two user defaults settings you're interested in changes. Then, the method changes the state of the corresponding checkbox to match the new value of the user defaults setting.

The code in the notification method is nearly identical to the code you just wrote in the `-windowDidBecomeKey:` method. The only difference is that the user defaults database is passed into the notification method as the notification's object, so you use it directly here by calling `-object` instead of calling the `NSUserDefaults +standardUserDefaults` class method as you did in `-windowDidBecomeKey:`.

Next, you should normally remove the observer before the preferences window controller is deallocated. This isn't actually necessary here because the preferences window controller is a singleton class and is not in fact deallocated during the life of the application. In case you change this in a future version of Vermont Recipes, go ahead and implement the `-dealloc` method. Insert it at the beginning of the `PreferencesWindowController.m` implementation file, like this:

```
- (void)dealloc {
    NSNotificationCenter *defaultCenter =
        [NSNotificationCenter defaultCenter];
    if (floor(NSFoundationVersionNumber)
        <= NSFoundationVersionNumber10_5) {
        [defaultCenter removeObserver:self];
    } else {
        [defaultCenter removeObserver:
            [self userDefaultsDidChangeObserver]];
    }
    [super dealloc];
}
```

You should not remove the observer by implementing the `-windowWillClose:` delegate method even if the controller were not a singleton class. The reason should be clear from the previous discussion of `-awakeFromNib` and `-windowDidLoad`. If you remove the observer when the window closes, the observer will no longer exist when the user reopens the window because `-windowDidLoad`, where the observers are registered, is not called in that situation.

As you did once before, you can use a runtime test instead of a preprocessor macro to detect whether Leopard or Snow Leopard is running because the `-removeObserver:` method is available in both versions of Mac OS X.

The Snow Leopard versions of the registration and removal code call accessor methods to get and set the observer. Set up the instance variable and accessor methods

next. In the `PreferencesWindowController.h` header file, declare the instance variable at the beginning of the curly braces block in the `@interface` directive like this:

```
id userDefaultsDidChangeObserver;
```

Declare its accessor methods at the beginning of the Accessor Methods section of the header file like this:

```
- (void)setUserDefaultsDidChangeObserver:(id)observer;  
- (id)userDefaultsDidChangeObserver;
```

Define them at the beginning of the Accessor Methods section of the implementation file like this:

```
- (void)setUserDefaultsDidChangeObserver:(id)observer {  
    userDefaultsDidChangeObserver = observer;  
}  
  
- (id)userDefaultsDidChangeObserver {  
    return userDefaultsDidChangeObserver;  
}
```

Now the two checkboxes in the General pane of the preferences window are updated not only when the user opens the preferences window but also every time the user selects or deselects the suppression checkbox in a warning alert. The effect doesn't actually take place until the user dismisses the alert, because that is when the user commits the decision to suppress the alert. The user defaults database is updated and `NSUserDefaults` posts the `NSUserDefaultsDidChangeNotification` notification.

10. You have now solved the problem of updating a checkbox in the General pane of the preferences window to match the user's change to a suppression button in the warning alert, but you should also deal with the reverse situation. When the user changes a checkbox in the General pane, the corresponding suppression button in a warning alert should be updated. This is not an issue with respect to the `VRDefaultSuppressScaleUpAlertKey` setting, because the alert containing that suppression checkbox is application modal, and the computer beeps at you only if you try to click a checkbox in the preferences window while the alert is open. The alert containing the `VRDefaultSuppressRestoreDiaryDocumentAlertKey` checkbox, however, is only document modal. It is therefore possible for the user to open the preferences window and change the corresponding checkbox in the General pane while the alert is open. It would be nice if the suppression checkbox in the alert changed to match any change to the checkbox in the General preferences pane, because user changes in the preferences window should always take effect immediately.

Save a reference to the warning alert when the diary window controller displays it, and register to observe the `NSUserDefaultsDidChangeNotification` notification

there. This is the mirror image of the technique you used to update the preferences window when the user changes the state of the suppression button in the alert.

Start by setting up an instance variable and accessor methods to capture the alert. In the `DiaryWindowController.h` header file, add this instance variable at the end of the `@interface` directive:

```
NSAlert *restoreAutosavedDiaryDocumentAlert;
```

Declare the accessor methods at the end of the Accessor Methods section in the header file like this:

```
- (void)setRestoreAutosavedDiaryDocumentAlert:(NSAlert *)alert;
- (NSAlert *)restoreAutosavedDiaryDocumentAlert;
```

Define the accessor methods at the end of the Accessor Methods section in the implementation file like this:

```
- (void)setRestoreAutosavedDiaryDocumentAlert:(NSAlert *)alert {
    restoreAutosavedDiaryDocumentAlert = alert;
}

- (NSAlert *)restoreAutosavedDiaryDocumentAlert {
    return restoreAutosavedDiaryDocumentAlert;
}
```

The rest of the new code goes in the Alerts section at the end of the `DiaryWindowController.m` implementation file. First, add these statements to the existing `-alertDidRestoreAutosavedDiaryDocument` method, just before the `return alert` statement:

```
[self setRestoreAutosavedDiaryDocumentAlert:alert];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(userDefaultsDidChange:)
    name:NSUserDefaultsDidChangeNotification object:nil];
```

Next, add this statement at the end of the existing `-alertDidRestoreAutosavedDiaryDocumentDidEnd:returnValue:contextInfo:` method:

```
[[NSNotificationCenter defaultCenter] removeObserver:self
    name:NSUserDefaultsDidChangeNotification object:nil];
```

Finally, add the notification method following the callback method, like this:

```
- (void)userDefaultsDidChange:(NSNotification *)notification {
    [[[self restoreAutosavedDiaryDocumentAlert]
        suppressionButton] setState:[notification object]
        forKey:VRDefaultSuppressRestoreDiaryDocumentAlertKey]];
}
```

That's all there is to it. The code you added to the `-alertDidRestoreAutosavedDiaryDocument` method registered the diary window controller as an observer for the same `NSUserDefaultsDidChangeNotification` notification that you have been using throughout this step. The code you added to the callback method simply removed the diary window controller as an observer, because observing the notification is only necessary while the warning alert is being displayed. The notification method, `-userDefaultsDidChange:`, used the new `-restoreAutosavedDiaryDocumentAlert` accessor method to get the alert and set the state of its suppression button to match the changed state of the corresponding checkbox in the General pane of the preferences window. It did this by getting the user defaults database from the notification argument's object and reading the current state of the setting of interest.

To avoid unnecessary repetition, I did not spell out the Snow Leopard variants of the notification observer registration and removal methods or their supporting methods. They are laid out earlier in this step for the mirror-image operation, and they are included in the downloadable project file for Recipe 8.

The notification technique used here should be future proof, because `NSAlert`'s `-suppressionButton` method is an officially supported part of the Cocoa frameworks. It should continue to work even if Apple does move the suppression button around within the alert in a future release of Mac OS X.

11. You can now build and run the application and test the interaction between the checkboxes in the General pane of the preferences window and the two warning alerts.

The setting relating to the alert that warns about print scaling over 100% is easiest to test. Create or open the Chef's Diary, and then choose `File > Print`. Next, choose Preferences from the application menu, and move the preferences window so that it will remain visible while you try to print a Chef's Diary document. You can't choose preferences after you have triggered this alert because it is application modal and the Preferences menu item would be disabled. Now try to enter a number greater than 100% in the Scale field of the Print panel. If you haven't previously suppressed it, an alert is immediately displayed, warning you that you cannot print larger than 100%. Try selecting one of the checkboxes in the General pane of the preferences window, and the computer just beeps because the alert is application modal. Now select the suppression checkbox in the alert. The corresponding checkbox in the General pane of the preferences window does not change, because you haven't yet committed to the new setting of the suppression checkbox. Now dismiss the alert, and the checkbox in the General pane immediately becomes checked even though the preferences window is behind the Chef's Diary window and its Print panel. Try to enter a number greater than 100% in the Scale field again, and no alert appears. Deselect

the checkbox in the General pane and try to enter a number greater than 100% in the Scale field again. This time, the alert appears. Dismiss the alert, select the checkbox in the General pane, and close the preferences window; then choose Preferences from the application menu to reopen it. The General pane reopens with the checkbox selected, as it should. Deselect both checkboxes in the preference pane, close the window, dismiss the print dialog, and quit the application.

Now test the other alert, which warns that you have reopened an autosaved document. This is more complicated because you have to crash the application and relaunch it to trigger the alert. Launch Vermont Recipes, create or open the Chef's Diary, type a word or two, and let it sit for more than 5 seconds to allow autosave to kick in. Then click the red Tasks button in any Xcode window to kill the application. Relaunch it, and the alert appears, informing you that the document "Untitled" was restored from an autosaved copy. Choose Preferences in the application menu, and move the preferences window so that you can see it and the alert at the same time. You were able to open the preferences window while the alert was open because this alert is only document modal. In the General pane, select the "Don't show alert when restoring from autosaved document" checkbox, and the suppression button in the alert is immediately selected. Deselect the checkbox in the General pane, and the suppression button in the alert is immediately deselected. Now select the suppression button in the alert. The corresponding checkbox in the General pane remains unchanged. Dismiss the alert, and the checkbox in the General pane is deselected because you committed the changed value of the suppression button.

Step 4: Configure the Recipes Tab View Item

The Recipes tab view item contains several user interface elements that govern a single feature of the recipes window: its standard state. Recall from Recipe 7 that the *standard state* of a window is its initial size as set in the Window Attributes inspector. This is the size it assumes when the user first opens it. Thereafter, the user sets the window's current *user state* by dragging the window's resize control to resize it. When the user clicks the window's zoom button, the window toggles between its standard state and its user state.

The Recipes tab view item allows the user to change the window's standard state. After setting a new standard state, zooming the window causes it to toggle between its current user state and the new standard state. This may be useful to any user

who has a very small display or a very large display and is unhappy with the default standard state as set in Interface Builder.

The Recipes Tab View item provides three different ways to change the standard state: width and height text fields where the user can enter specific values, steppers where the user can increment and decrement the width and height, and a button that the user can click to set the width and height to the current size of the recipes window. If the recipes window is open, its size changes immediately when the user changes the width or height using its text field or stepper. This is consistent with the principle that changes to preference settings should take effect immediately, and it is useful because it allows the user to visualize the effect of every new setting. The button to set the window's standard state to its current size is disabled when the recipes window is not open.

1. Start by writing IBOutlet instance variables and accessor methods for the text fields and steppers. You will need outlets to set the values of the text fields and steppers based on their associated user defaults values and when the user changes them. The four outlets require instance variable and accessor declarations in the `PreferencesWindowController.h` header file and definitions in the implementation file. They are standard boilerplate, so look them up in the downloadable project file for Recipe 10.

Connect all of the outlets in the `PreferencesWindow` nib file by Control-dragging from the File's Owner proxy to a text field or stepper and choosing its outlet.

2. Next, arrange to display the current standard state of the recipes window in the width and height fields as soon as the user opens the preferences window. In order to do this, you must of course first obtain the current standard state.

In Recipe 7, you set the recipes window's standard state in the Window Size inspector in Interface Builder by setting the Content Frame width and height to 1200 by 800 pixels. The window's size is in fact 1200 by 878 pixels, larger than the content frame, because the content frame excludes the window's title bar and toolbar. In the `-windowDidLoad` method in the `RecipesWindowController.m` implementation file, you got the window's frame by calling `NSWindow`'s `-frame` method, and you set the user defaults value for the `VRDefaultRecipesWindowStandardSizeKey` using the frame's size member. This worked because the window had just been read in from the nib file, and you did not anticipate needing to know its standard state earlier than that.

Now, however, you need to get the recipes window's size for display in the preferences window, and the user might not yet have opened the recipes window. You don't want to load the `RecipesWindow` nib file prematurely just to get its standard state for display in the preferences window. Instead, set the standard state programmatically in the initial user defaults, using the same figures you used to set the content frame in the nib file. You can leave the nib file as it is,

because you won't use its content frame settings at all, relying instead on the coded initial user defaults values.

You already wrote an `+initialize` method in Recipe 9, so you know how to do this. There, you set the initial user defaults values for the diary document's printing content flags, such as the value keyed to `VRDefaultDiaryPrintTagsKey`. Here, you need to do exactly the same thing for the recipes window's initial standard state, keyed to `VRDefaultRecipesWindowStandardSizeKey`. You can't place this `+initialize` method in the recipes window controller, however, because the user might open the preferences window before opening the recipes window. And you can't place it in the preferences window controller, because the user might open the recipes window before opening the preferences window. To make sure that the recipes window's standard state gets set up in the user defaults before it is needed, place the `+initialize` method in the application controller.

Insert this method at the beginning of the `VRApplicationController.m` implementation file after the `@implementation` directive:

```
#pragma mark INITIALIZATION
```

```
+ (void)initialize {
    if (self == [VRApplicationController class]) {
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        NSDictionary *initialUserDefaults =
            [NSDictionary dictionaryWithObjectsAndKeys:
             NSStringFromSize(NSMakeSize(1200.0, 800.0)),
             VRDefaultRecipesWindowStandardSizeKey, nil];
        [defaults registerDefaults:initialUserDefaults];
    }
}
```

Recall that you have been saving the recipes window's standard size in the user defaults as a string instead of a number.

Because `VRDefaultRecipesWindowStandardSizeKey` is declared in the recipes window controller, you must import the controller's header file at the top of the `VRApplicationController.m` implementation file, like this:

```
#import "RecipesWindowController.h"
```

Before moving on, revise the `-windowDidLoad` method in the `RecipesWindowController.m` implementation file so that it uses the initial user defaults value for the recipes window's standard state, instead of reading the standard state from the nib file. Replace the three statements that get the standard size by reading the recipes window's frame and saving it to the user defaults with the statements

shown next. They get the standard size from the user defaults and resize the window accordingly.

```
NSSize defaultWindowSize =
    NSSizeFromString([[NSUserDefaults standardUserDefaults]
        stringForKey:VRDefaultRecipesWindowStandardSizeKey]);
NSRect designedWindowFrame = [[self window] frame];
NSRect newWindowFrame = NSMakeRect(designedWindowFrame.origin.x,
    designedWindowFrame.origin.y + (designedWindowFrame.size.height
    - defaultWindowSize.height), defaultWindowSize.width,
    defaultWindowSize.height);
[[self window] setFrame:newWindowFrame display:YES];
```

This code takes into account the fact that the standard size of the window as you just set it in the initial user defaults differs from the size of the window as it is loaded from the nib file. Because the window's origin is at the bottom-left corner, you adjust the origin to take account of the difference in height so that the window's position measured at the top-left corner remains the same. It still uses the window's horizontal coordinates as derived from the nib file, in order to take advantage of the nib loading mechanism for centering the window horizontally.

3. Now that you've set the recipes window's standard state in the user defaults, you can display the width and height when the user opens the preferences window. Add these statements at the end of the `-windowDidLoad` method in the `PreferencesWindowController.m` implementation file:

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
NSSize recipesWindowSize = NSSizeFromString([defaults
    stringForKey:VRDefaultRecipesWindowStandardSizeKey]);
[[self recipesWindowWidthTextField] setIntegerValue:
    (NSInteger)recipesWindowSize.width];
[[self recipesWindowHeightTextField] setIntegerValue:
    (NSInteger)recipesWindowSize.height];
[[self recipesWindowWidthStepper] setIntegerValue:
    (NSInteger)recipesWindowSize.width];
[[self recipesWindowHeightStepper] setIntegerValue:
    (NSInteger)recipesWindowSize.height];
```

You have already configured the number formatter for the width and height text fields to handle integer values. You therefore cast the width and height members of the size member you obtained from the user defaults, which are `CGFloat` values, to `NSInteger`, discarding the fractional part. You then set the integer values of the width and height text fields and the steppers accordingly. The number

formatters attached to the text fields automatically format and display them in accordance with your formatter configuration settings.

Because `VRDefaultRecipesWindowStandardSizeKey` is declared in the recipes window controller, you must import the controller's header file at the top of the `PreferencesWindowController.m` implementation file, just as you did a moment ago in the `VRApplicationController.m` implementation file, like this:

```
#import "RecipesWindowController.h"
```

4. Finally, you get to the point of this exercise, which is to allow the user to change the recipes window's standard state. To do this, you need action methods connected to the two text fields and the two steppers. You also need an action method for the Use Current Size button, but you'll deal with that after you have the text fields and steppers working.

It turns out that you can connect a single action method both to the width text field and the width stepper, and another action method both to the height text field and the height stepper. Write the width action method first; the height action method is essentially identical. Declare the action method at the end of the Action Methods section in the `PreferencesWindowController.h` header file, like this:

```
- (IBAction)setDefaultRecipesWindowWidth:(id)sender;
```

Define it in the implementation file like this:

```
- (IBAction)setDefaultRecipesWindowWidth:(id)sender {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSSize standardSize = NSSizeFromString([defaults
        stringForKey:VRDefaultRecipesWindowStandardSizeKey]);
    standardSize.width = (CGFloat)[sender integerValue];
    [defaults setObject:NSStringFromSize(standardSize
        forKey:VRDefaultRecipesWindowStandardSizeKey);

    if ([sender isKindOfClass:[NSStepper class]]) {
        [[self recipesWindowWidthTextField] takeIntegerValueFrom:sender];
    } else {
        [[self recipesWindowWidthStepper] takeIntegerValueFrom:sender];
    }

    if ([[sender window] isZoomed]) {
        NSWindow *recipesWindow = [[[[[VRDocumentController
            sharedDocumentController] recipesDocument]
            windowControllers] objectAtIndex:0] window];
```

(code continues on next page)

```

        NSRect newWindowFrame = NSMakeRect([recipesWindow frame].origin.x,
        [recipesWindow frame].origin.y, standardSize.width,
        standardSize.height);
        [recipesWindow setFrame:newWindowFrame display:YES];
    }
}

```

The first block of statements gets the standard size from the user defaults, sets its width member to the NSInteger value that the user entered in the width field, cast to a CGFloat, and writes the standard size back out to the user defaults. Recall that this action method will be connected both to the width text field and the width stepper. The **sender** argument is thus either the text field or the stepper. Both have a value, and as long as the values are kept synchronized, it doesn't matter which you use to update the user defaults.

The second block of statements takes care of synchronizing the width text field and the width stepper. If the **sender** is the stepper because the user used the stepper to increment or decrement the width, the code sends NSControl's **-takeIntegerValueFrom:** message to the text field, telling it to take its value from the stepper. If the **sender** is the text field because the user just typed a value into it, the code sends the same message to the stepper, telling it to take its value from the text field. Using **-takeIntegerValueFrom:** illustrates how you can synchronize related controls without using the raw data. In this case, however, since the **standardSize** local variable holds the data, it might be cleaner to set both controls directly using **-setIntegerValue: standardSize.width**.

The third block of statements actually changes the width of the recipes window, if it is open, to match the new value set by the user. This gives the application a very "live" feel, especially if the user holds down a stepper arrow, causing the window's width to ratchet upward by 10-pixel increments continuously. You could have achieved this effect by using the same **-userDefaultsDidChange:** notification method that you used in Step 3 to update the checkboxes in the General pane. Here, however, you do not anticipate that any external actor will change the standard recipes window state, so you change it directly by messaging the recipes window to set its frame to the new value and display it.

You referred to VRDocumentController in the last block. Be sure to import its header by adding this line at the top of the PreferencesWindowController.m implementation file:

```
#import "VRDocumentController.h"
```

The reason you referred to VRDocumentController was to call its **-recipesWindow** method, which doesn't yet exist. Add it to the VRDocumentController.h header

file now, inserting this declaration immediately before the existing `-diaryDocument` method that you wrote in Recipe 6:

```
- (NSDocument *)recipesDocument;
```

Define it in the `VRDocumentController.m` implementation file on the model of the existing `-diaryDocument` method, like this:

```
- (NSDocument *)recipesDocument {
    for (NSDocument *thisDocument in [self documents]) {
        if ([thisDocument isKindOfClass:[RecipesDocument class]]) {
            return thisDocument;
        }
    }
    return nil;
}
```

You must also import the `RecipesDocument.h` header into the `VRDocumentController.m` implementation file, like this:

```
#import "RecipesDocument.h"
```

Now go back to the Action Methods section of the `PreferencesWindowController` header and implementation file and add the action method for the height text field and the height stepper, `-setDefaultRecipesWindowHeight:`. It is almost identical to the width action method, so look it up in the downloadable project file for Recipe 10.

Don't forget to connect the two action methods in Interface Builder. In the `PreferencesWindow` nib file, Control-drag from the width text field to the First Responder proxy and choose the corresponding action method. Do the same thing from the width stepper, since both the text field and the stepper are to use the same action method. Repeat the process with the height text field and the height stepper.

5. Now you can implement the Use Current Size button's action method. Look up `-setDefaultRecipesWindowSizeFromCurrent:` in the downloadable project file for Recipe 8.

This is essentially identical to the width and height action methods, except it does not have to change the size of the window because it uses the window's current size.

Don't forget to connect this action method in Interface Builder.

6. The Use Current Size button does present one complication, however. It should be disabled when the recipes window is not open. You learned all about user interface item validation in Recipe 4 when you validated a number of controls in the Chef’s Diary window. Use the same technique here.

Remember that user interface item validation involved declaring two protocols, `VRValidatedControl` and `VRControlValidations`, plus one or more validated control subclasses that conform to the `VRValidatedControl` protocol. In Step 4, you declared the two protocols in the `DiaryWindowController.h` header file, because that’s where you were using them. It was a bit shortsighted to place them there, because they would undoubtedly prove useful in other windows, as well. Now you need them for the preferences window.

Create a new file for them. In Xcode, choose `File > New File`. In the New File dialog, select `Cocoa Class` in the source list and “Objective-C protocol” in the top-right pane, and then click `Next`. In the next dialog, enter **`ValidationProtocols.h`** as the name of the file. Xcode is smart enough to know that protocols don’t have an implementation part, so you aren’t asked whether to create one. Make sure both the Vermont Recipes and Vermont Recipes SL targets are selected, and click `Finish`. Create a new group in the Groups & Files pane of the Vermont Recipes project window and name it **`Protocols`**. Then move the new `ValidationProtocols.h` header file into it. Add your standard file information at the top of the header file, and copy and paste the `VRValidatedControl` and `VRControlValidations` declarations from the `DiaryWindowController.h` header file exactly as they appeared when you wrote them in Recipe 4.

Remove the protocol declarations from the diary window controller. To make them usable there, import the new protocol header file in their place. At the top of the `DiaryWindowController.h` header file, add this line:

```
#import "ValidationProtocols.h"
```

You must import it in the `DiaryWindowController.h` header file because the three validated control subclasses you declare there—`ValidatedDiaryButton`, `ValidatedDiaryDatePicker`, and `ValidatedDiarySearchField`—all declare that they conform to the `VRValidatedControl` protocol.

Now go back to the `PreferencesWindowController.h` header file. Import the new `ValidationProtocols.h` header file there too, using the same `#import` preprocessor directive. Also, revise the `@interface` directive so that it declares conformance to the `NSUserInterfaceValidations` protocol, like this:

```
@interface PreferencesWindowController
    : NSWindowController <NSUserInterfaceValidations> {
```

Then, at the end of the header file, add this declaration of an NSButton subclass that conforms to the VRValidatedControl protocol:

```
#pragma mark -
```

```
@interface ValidatedPreferencesButton : NSButton <VRValidatedControl> {  
}
```

```
@end
```

Define it at the end of the PreferencesWindowController.m implementation file, like this:

```
#pragma mark -
```

```
@implementation ValidatedPreferencesButton
```

```
- (void)validate {  
    id validator = [NSApp targetForAction:[self action] to:[self target]  
                  from:self];  
    if ((validator == nil)  
        || ![validator respondsToSelector:[self action]]) {  
        [self setEnabled:NO];  
    } else if ([validator respondsToSelector:  
                @selector(validateControl:)]) {  
        [self setEnabled:[validator validateControl:self]];  
    } else if ([validator respondsToSelector:  
                @selector(validateUserInterfaceItem:)]) {  
        [self setEnabled:[validator validateUserInterfaceItem:self]];  
    } else {  
        [self setEnabled:YES];  
    }  
}
```

```
@end
```

This is identical to the validated control subclasses you wrote in Recipe 4.

To cause the Use Current Size button to be validated, you must turn it into a ValidatedPreferencesButton. Do this by selecting the button in the nib file's design surface; then open the Button Identity inspector and choose Validated-PreferencesButton from the Class combo box. Select the General pane in the preferences window design surface and save the nib file.

Finally, add the `-validateUserInterfaceItem:` method and the related `-updateWindow` method. At the end of the `PreferencesWindowController` class interface in the `PreferencesWindowController.h` header file, declare the `-updateWindow` method like this:

```
#pragma mark USER INTERFACE VALIDATION
```

```
- (void)updateWindow;
```

At the end of the `PreferencesWindowController` class interface in the `PreferencesWindowController.m` implementation file, define both methods like this:

```
#pragma mark USER INTERFACE VALIDATION
```

```
- (void)updateWindow {
    for (id thisTabViewItem in [(NSTabView *)[self window]
        initialFirstResponder] tabViewItems]) {
        for (id thisView in [[thisTabViewItem view] subviews]) {
            if ([thisView conformsToProtocol:
                @protocol(VRValidatedControl)]) {
                [thisView validate];
            }
        }
    }
}

- (BOOL)validateUserInterfaceItem:
    (id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if (action == @selector(setDefaultRecipesWindowSizeFromCurrent:))
    {
        NSWindow *recipesWindow = [[[[[VRDocumentController
            sharedDocumentController] recipesDocument]
            windowControllers] objectAtIndex:0] window];
        return (recipesWindow != nil);
    }
    return YES;
}
```

These are very similar to the corresponding methods you wrote in Recipe 4. The `-updateWindow` method is a little different because it contains a tab view. It must loop through each tab view item and then loop through all the subviews in each tab view item's view.

One final requirement is to implement the `-windowDidUpdate:` delegate method that calls `-updateWindow`. At the end of the Delegate Methods section of the `PreferencesWindowController.m` implementation file, add this method:

```
- (void)windowDidUpdate:(NSNotification *)notification {  
    [self updateWindow];  
}
```

7. You're ready to test the Recipes pane of the preferences window. First move the Vermont Recipes preferences file to the Trash to start with a clean slate.

Build and run the application. Then choose Preferences from the application menu and select the Recipes tab in the preferences window. The width and height fields show that the recipes window's standard size is 1200 by 800 pixels. The recipes window opened automatically when you launched Vermont Recipes, so you can see that it is in fact about that size. Move the preferences window to one size so that you can see it and the recipes window at the same time.

The Use Current Size button is enabled. Close the recipes window, and you see that the button becomes disabled. Choose File > New Recipes File and bring the preferences window to the front. The button is once again enabled.

Type a new width in the width field, say, **800** pixels. The moment you press Enter, Return, or Tab or click out of the width field, the recipes window changes so that it is only 800 pixels wide.

Hold down the down arrow in the width stepper. The recipes window grows progressively narrower.

Bring the recipes window to the front and click the zoom button repeatedly. The recipes window toggles between its former user state and the new, narrower standard state.

Drag the recipes window's resize control to make the window as small as possible. Click the zoom button repeatedly again, and the window toggles between the new, very small user state and the new, narrower standard state.

Repeat these tests with the height text field and stepper.

Finally, use the recipes window's resize control to make the window very large. Click the Use Current Size button. The values displayed in the width and height fields change to reflect the window's current dimensions. Click the recipes window's zoom button repeatedly, and nothing happens because the user state and the standard state are now identical. Resize the recipes window and click the zoom button repeatedly again. Now the window toggles between its new user state and the new, very large standard state.

Close the recipes window and reopen it. It reopens at its new standard state.

Step 5: Configure the Chef's Diary Tab View Item

The Chef's Diary tab view item contains four sections. Two of them, at the top, duplicate functionality you have already implemented for other views. I will outline what you need to do to implement them here, leaving most of the work as an exercise for the reader. In both cases, the code and Interface Builder settings are fully implemented in the downloadable project file for this recipe.

1. The first section of the Chef's Diary pane is a group of controls that are identical to the controls that you just implemented in the Recipes pane. You should implement the Chef's Diary controls exactly the same way you implemented the corresponding controls in the Recipes pane, but use the diary window controller instead of the recipes window controller. I won't repeat the instructions here, but they are fully implemented in the downloadable project file for this recipe. Be sure to follow all of the tasks in Step 4, including using Interface Builder to make the necessary outlet and action connections and to reset the class of the Use Current Size button. In addition to adding outlet accessors and action methods, there are several methods you wrote in Step 4 that must be modified to work with the diary window as well as the recipes window. For example, set the initial standard size of the diary window in `+VRApplicationController initialize` to 600.0 by 900.0 pixels by adding another object and the `VRDefaultDiaryWindowStandardSizeKey` key to the call to `+dictionaryWithObjectsAndKeys:`.
2. The controls in the Printing section of the Chef's Diary pane are identical to the corresponding controls in the All Print Jobs section of the DiaryPrint-PanelAccessoryView nib file that you created in Recipe 9.

Their accessor methods are identical to those you wrote in Recipe 9 as well. Simply copy and paste the declarations and the implementations of the `-printTagsCheckbox`, `-printHeadersAndFootersCheckbox`, and `-printTimestampRadioGroup` accessor methods from the DiaryPrintPanelAccessoryController header and implementation files into the PreferencesWindowController header and implementation files, and connect all of them in the PreferencesWindow nib file. Copy and paste the instance variables corresponding to the accessor methods.

The action methods are different. In the diary Print panel accessory controller, the action methods set the printing defaults through a represented object. Here, you set the defaults directly, just as you did in the action methods for the

General and Recipes panes. Declare them in the PreferencesWindowController.h header file like this:

- (IBAction)setDefaultDiaryPrintTags:(id)sender;
- (IBAction)setDefaultDiaryPrintHeadersAndFooters:(id)sender;
- (IBAction)setDefaultDiaryPrintTimestamp:(id)sender;

Define them in the PreferencesWindowController.m implementation file like this:

- (IBAction)setDefaultDiaryPrintTags:(id)sender {
 [[NSUserDefaults standardUserDefaults] setBool:[sender state]
 forKey:VRDefaultDiaryPrintTagsKey];
}
- (IBAction)setDefaultDiaryPrintHeadersAndFooters:(id)sender {
 [[NSUserDefaults standardUserDefaults] setBool:[sender state]
 forKey:VRDefaultDiaryPrintHeadersAndFootersKey];
}
- (IBAction)setDefaultDiaryPrintTimestamp:(id)sender {
 [[NSUserDefaults standardUserDefaults] setInteger:[sender selectedRow]
 forKey:VRDefaultDiaryPrintTimestampKey];
}

You must import the DiaryPrintPanelAccessoryController.h header file into the PreferencesWindowController.m implementation file in order to use the keys for the user defaults printing settings, like this:

```
#import "DiaryPrintPanelAccessoryController.h"
```

Connect the action methods in Interface Builder.

To display the user defaults printing settings when the user opens the preferences window, you must first set them in the preferences window controller's `+initialize` method, in the PreferencesWindowController.m implementation file. You wrote the necessary `+initialize` method in Recipe 9, but you placed it in the DiaryDocument.m implementation file. That is too late if the user opens the preferences window before opening the Chef's Diary. Delete the `+initialize` method from the diary document implementation file, and instead initialize its objects with the appropriate keys in the `+ [VRApplicationController initialize]` method you started writing in Step 4. It should now look like this:

```
+ (void)initialize {  
    if (self == [VRApplicationController class]) {  
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
```

(code continues on next page)

```

        NSDictionary *initialUserDefaults = [NSDictionary
            dictionaryWithObjectsAndKeys:
                NSStringFromSize(NSMakeSize(1200.0, 800.0)),
                VRDefaultRecipesWindowStandardSizeKey,
                NSStringFromSize(NSMakeSize(600.0, 900.0)),
                VRDefaultDiaryWindowStandardSizeKey,
                [NSNumber numberWithInt:NO],
                VRDefaultDiaryPrintTagsKey,
                [NSNumber numberWithInt:YES],
                VRDefaultDiaryPrintHeadersAndFootersKey,
                [NSNumber numberWithInt:0],
                VRDefaultDiaryPrintTimestampKey, nil];
        [defaults registerDefaults:initialUserDefaults];
    }
}

```

You could leave out the settings for `NO` and `0` because `NSUserDefaults` defaults to those values, but I find my code easier to maintain if I initialize all related defaults explicitly.

Don't forget to import the `DiaryPrintPanelAccessoryController.h` header file into the `VRApplicationController.m` implementation file, in order to use the printing defaults keys, like this:

```
#import "DiaryPrintPanelAccessoryController.h"
```

Now add statements at the end of the `-windowDidBecomeKey:` method in the `PreferencesWindowController.m` implementation file to display the printing user defaults when the user opens the preferences window. The printing controls, like the General pane checkboxes, have to be displayed in the `-windowDidBecomeKey:` method instead of the `-windowDidLoad` method because the printing controls can be changed externally, in the custom accessory view of the Chef's Diary Print panel. Here are the statements to add:

```

[[self printTagsCheckbox] setState:([defaults boolForKey:
    VRDefaultDiaryPrintTagsKey]) ? NSOnState : NSOffState];
[[self printHeadersAndFootersCheckbox] setState:([defaults boolForKey:
    VRDefaultDiaryPrintHeadersAndFootersKey]) ? NSOnState : NSOffState];
[[self printTimestampRadioGroup] setState:NSOnState atRow:
    [defaults integerForKey:VRDefaultDiaryPrintTimestampKey] column:0];

```

The printing section of the Chef's Diary pane in the preferences window is now working, except for one issue. In Step 3, which also involved user defaults settings that could be changed in two places, you arranged to observe the `NSUserDefaultsDidChangeNotification` notification so that a change in one

place would be immediately reflected onscreen in the other place. For the sake of consistency and a good user experience, you should implement the same behavior here in case the user has the Chef's Diary Print panel and the Chef's Diary pane of the preferences window open at the same time. I won't repeat in full the explanation of how to synchronize these settings—this is left as an exercise for the reader. Just follow tasks 9 and 10 of Step 3 with appropriate changes. The necessary code is in place in the downloadable project file for Recipe 10.

Here's a hint for updating the preferences window when the user makes changes in the Print panel and dismisses it: The preferences window controller is already registered to observe the `NSUserDefaultsDidChangeNotification` notification, and you have already written the code to respond to the notification and to remove the observer. All you have to do to update the Chef's Diary pane when the user changes the Print panel and dismisses it is to add three statements to the existing `-userDefaultsDidChange:` method. In fact, you've already written them, so this requires nothing more than copying and pasting the statements or writing a method. They're in the `-windowDidBecomeKey:` method.

Updating the Print panel when the user changes settings in the Chef's Diary pane will take almost as little work. Most of the code that you wrote in the preferences window controller to update the preferences window when the user changes settings in the Print panel can be copied and pasted into the diary Print panel accessory controller, because the instance variables for the accessor methods for the two checkboxes and the radio group, as well as for the `userDefaultsDidChangeObserver` notification observer, are named the same in both files. Copy and paste the following declarations and implementations from the `PreferencesWindowController` class into the `DiaryPrintPanelAccessoryController` class: the `userDefaultsDidChangeObserver` instance variable; the `-setUserDefaultsDidChangeObserver:` and `-userDefaultsDidChangeObserver` accessor methods; and the `-dealloc` method to unregister the observer.

There are only two places where you need to write new code, and even it has already been written. At the end of the `-loadView` method in the `DiaryPrintPanelAccessoryController.m` implementation file, add these statements to register the observer—you can copy them from the `-windowDidLoad` method in the `PreferencesWindowController.m` implementation file and paste them into the `-loadView` method:

```
NSNotificationCenter *defaultCenter =
    [NSNotificationCenter defaultCenter];
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_6
[defaultCenter addObserver:self selector:
    @selector(userDefaultsDidChange:)
    name:NSUserDefaultsDidChangeNotification object:nil];
```

(code continues on next page)

```

#else
    [self setUserDefaultsDidChangeObserver:[defaultCenter
        addObserverForName:NSUserDefaultsDidChangeNotification
        object:nil queue:nil
        usingBlock:^(NSNotification *notification) {
            [self userDefaultsDidChange:notification];
        }]];
#endif

```

The other method you have to write is the notification method itself, to be added after the Override Methods section of the `DiaryPrintPanelAccessoryController.m` implementation file:

```

#pragma mark NOTIFICATION METHODS

- (void)userDefaultsDidChange:(NSNotification *)notification {
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    [[self printTagsCheckbox] setState:([defaults
        boolForKey:VRDefaultDiaryPrintTagsKey])
        ? NSOnState : NSOffState];
    [[self printHeadersAndFootersCheckbox] setState:([defaults
        boolForKey:VRDefaultDiaryPrintHeadersAndFootersKey])
        ? NSOnState : NSOffState];
    [[self printTimestampRadioGroup] setState:NSOnState
        atRow:[defaults integerForKey:VRDefaultDiaryPrintTimestampKey]
        column:0];
}

```

Again, you can copy the body of this method from the corresponding method in the preferences window controller and paste it in. Declare it in the `DiaryPrintPanelAccessoryController.h` header file like this:

```

#pragma mark NOTIFICATION METHODS

- (void)userDefaultsDidChange:(NSNotification *)notification;

```

You're now done with the Printing section of the Chef's Diary pane. When you change these settings in the preferences window, the Print panel recognizes them either immediately, if it was open, or later when the user opens it. Changes to these settings in the Print panel are reflected in the preferences window, either when the user closes the Print panel to commit the changes, or later when the user opens the preferences window.

3. Next, implement the Autosaving section of the Chef's Diary pane of the preferences window. This is very straightforward.

In Recipe 7, you implemented the `-applicationDidfinishLaunching:` delegate method in `VRApplicationController` to set the autosaving delay to 5.0 seconds. This was a stopgap to turn on autosaving and set the interval to a very short period for purposes of testing. Now you should give the user the option to turn off autosaving by setting the interval to 0.0 as well as some more realistic intervals.

As with all user defaults settings, you need a key under which to store the value of the interval, and an action method to connect to the pop-up button in the Autosaving section of the Chef's Diary pane. You should also set the initial default value in the `+initialize` method in `VRApplicationController`, and you should arrange for the preferences window to display the current setting when the user opens it.

Start by defining the key. This preference is to apply to any and all documents, so a good place to define it is in the `VRDocumentController` class. That's where Cocoa declares the `-setAutosavingDelay:` method, too. Add this declaration before the `@interface` directive at the top of the `VRDocumentController.h` header file:

```
extern NSString *VRDefaultDiaryAutosaveIntervalKey;
```

Define it at the end of the `VRDocumentController.m` implementation file like this:

```
NSString *VRDefaultDiaryAutosaveIntervalKey =  
    @"diary document autosave interval";
```

Write the action method in the preferences window controller. Declare it at the end of the Action Methods section of the `PreferencesWindowController.h` header file:

```
- (IBAction)setDefaultDiaryAutosaveInterval:(id)sender;
```

Define it in the implementation file:

```
- (IBAction)setDefaultDiaryAutosaveInterval:(id)sender {  
    NSTimeInterval interval;  
    switch ([sender indexOfSelectedItem]) {  
        case 0:  
            interval = 15.0;  
            break;  
        case 1:  
            interval = 30.0;  
            break;  
        case 2:  
            interval = 60.0;  
            break;
```

(code continues on next page)

```

        case 3:
            interval = 300.0;
            break;
        case 4:
            interval = 0.0;
            break;
    }
    [[NSUserDefaults standardUserDefaults] setDouble:interval
        forKey:VRDefaultDiaryAutosaveIntervalKey];
}

```

The user defaults value is saved as a double because the `NSTimeInterval` type is declared as a double.

The `VRDocumentController.h` header file is already imported into the `PreferencesWindowController.m` implementation file, so the `VRDefaultDiaryAutosaveIntervalKey` key is available here.

Build the project and connect the action method to the First Responder proxy in the `PreferencesWindow` nib file in Interface Builder.

To display the preference setting when the user opens the preferences window, you need an instance variable and accessor method for the pop-up menu. Declare the instance variable and getter in the `PreferencesWindowController.h` header file separately like this:

```
IBOutlet NSPopUpButton *diaryAutosaveIntervalButton;
```

```
- (NSPopUpButton *)diaryAutosaveIntervalButton; // ADDED IN RECIPE 10
```

Define it in the `PreferencesWindowController.m` implementation file like this:

```

- (NSPopUpButton *)diaryAutosaveIntervalButton {
    return diaryAutosaveIntervalButton ;
}

```

Build the project and connect the outlet.

Now add the following code to the end of the `-windowDidLoad` method in the `PreferencesWindowController.m` implementation file:

```

double interval = [defaults doubleForKey:
    VRDefaultDiaryAutosaveIntervalKey];
NSInteger idx;
if (interval == 15.0) {
    idx = 0;
}

```



```

} else if (interval == 30.0) {
    idx = 1;
} else if (interval == 60.0) {
    idx = 2;
} else if (interval == 300.0) {
    idx = 3;
} else if (interval == 0.0) {
    idx = 4;
}
[[self diaryAutosaveIntervalButton ] selectItemAtIndex:idx];

```

Finally, revise the existing `-applicationDidFinishLaunching:` delegate method so that it sets the current autosaving interval according to the user's preference:

```

- (void)applicationDidFinishLaunching:(NSNotification *)notification {
    [[VRDocumentController sharedDocumentController]
        setAutosavingDelay:[NSUserDefaults standardUserDefaults]
        doubleForKey:VRDefaultDiaryAutosaveIntervalKey]];
}

```

You're done. Build and run the application, open the preferences window, and select the Chef's Diary pane. The selected pop-up menu item is *Never*, because you did not set an initial default value for the autosaving interval and it therefore defaults to 0.0 seconds. The code you just wrote therefore sets the index to 4, which is the menu item titled *Never*.

I don't feel safe without autosaving, so make one more change to the code. In the `+initialize` method in `VRApplicationController`, add the object `[NSNumber numberWithInt:30.0]` and the key `VRDefaultDiaryAutosaveIntervalKey` just before the `nil` at the end of the call to `+dictionaryWithObjectsAndKeys:`.

4. The last section of the Chef's Diary pane contains a text field where the user can set the current Chef's Diary document. When the user first opens the preferences window, this text field should display the full path to the current Chef's Diary document, if there is one, or be left blank if there is not. The user can type in the full path to another diary document, such as a backup, or the full path to any PDF file, to change the current Chef's Diary to the other file.

To display the file's path, you need an instance variable and getter for the text field. In the `PreferencesWindowController.h` header file, separately declare them like this:

```

IBOutlet NSTextField *currentDiaryDocumentTextField;

- (NSTextField *)currentDiaryDocumentTextField;

```

Define the getter in the PreferencesWindowController.m implementation file like this:

```
- (NSTextField *)currentDiaryDocumentTextField {  
    return currentDiaryDocumentTextField;  
}
```

Build the project and connect the outlet in the PreferencesWindow nib file.

Add the following statements to the end of the `-windowDidBecomeKey:` delegate method in the PreferencesWindowController.m implementation file to display the current Chef's Diary's path in the text field. This has to be done in the `-windowDidBecomeKey:` delegate method instead of the `-windowDidLoad` method because the current diary document can be set externally, when the user saves the Chef's Diary.

```
NSString *path = [[[VRDocumentController sharedDocumentController]  
    currentDiaryURL] path];  
if (path) [[self currentDiaryDocumentTextField] setStringValue:path];
```

You check whether the path is `nil` because it might be if the file is not there. Setting the string value of a text field raises an exception if the argument is `nil`.

Add the same two statements at the end of the `-userDefaultsDidChange:` notification method in the PreferencesWindowController.m implementation file. Now the text field will update immediately if the user saves a new current diary document, even if the preferences window remains in the background.

The last task is to write an action method that allows the user to enter a new path in the current diary text field and make the file at that path the new current Chef's Diary. Declare the action method at the end of the Action Methods section of the PreferencesWindowController.h header file like this:

```
- (IBAction)setDefaultCurrentDiaryDocument:(id)sender;
```

Define it like this at the end of the same section of the PreferencesWindowController.m implementation file:

```
- (IBAction)setDefaultCurrentDiaryDocument:(id)sender {  
    NSString *path = [sender stringValue];  
    BOOL isDirectory;  
    BOOL fileExists = [[NSFileManager defaultManager]  
        fileExistsAtPath:path isDirectory:&isDirectory];  
    if (fileExists & !isDirectory) {  
        NSString *type;
```

```

        BOOL success = [[NSWorkspace sharedWorkspace] getInfoForFile:path
                        application:NULL type:&type];
        if (success && ([type isEqualToString:@"vrdiary"]
                        || [type isEqualToString:@"rtf"])) {
            [[VRDocumentController sharedDocumentController]
             setCurrentDiaryURL:[NSURL fileURLWithPath:path]];
        } else {
            [[self alertWrongFileType] beginSheetModalForWindow:
             [self window] modalDelegate:nil didEndSelector:NULL
             contextInfo:NULL];
        }
    } else {
        [[self alertNoSuchFile] beginSheetModalForWindow:
         [self window] modalDelegate:nil didEndSelector:NULL
         contextInfo:NULL];
    }
}

```

Connect the action method in the PreferencesWindow nib file.

The action method uses both of Cocoa's file system classes, `NSFileManager` and `NSWorkspace`, to determine whether the path the user entered is valid. The `NSFileManager` method `-fileExistsAtPath:isDirectory:` is called to determine whether a file exists at that path and, if so, whether it is a directory. Neither the diary document nor a Rich Text Format (RTF) file that can be used as a diary document is expected to be a bundle or directory. If a flat file exists at the given path, the `NSWorkspace` method `-getInfoForFile:application:type:` is called to get its type, which is its file extension. You could just as well have called `[path pathExtension]`. If the type is either `@vrdiary` or `@rtf`, the existing `-setCurrentDiaryURL:` method in `VRDocumentController` is called to set the new user defaults value for the current Chef's Diary.

It is very easy to type an incorrect file path, whether due to typographical errors or mistaken understanding of the file's path. It would therefore be friendly to the user to provide some feedback about the reason for any error. As alternatives to displaying an alert, there are other user interface elements for entering path information that may be easier to use, but Vermont Recipes already supports using the Finder to make any file the current Chef's Diary simply by opening it. There is no need to make this preferences window setting more robust, so simply display an informative error if a problem arises.

Declare the two alerts at the end of the primary class in the PreferencesWindowController.h header file like this:

```
#pragma mark ALERTS
```

```
- (NSAlert *)alertNoSuchFile;  
- (NSAlert *)alertWrongFileType;
```

Define them at the end of the primary class in the PreferencesWindowController.m implementation file like this:

```
#pragma mark ALERTS
```

```
- (NSAlert *)alertNoSuchFile {  
    NSBeep();  
    NSAlert *alert = [[[NSAlert alloc] init] autorelease];  
    [alert setMessageText:NSString(@"The file was not found.",  
        @"message text for NoSuchFile alert")];  
    [alert setInformativeText:[NSString stringWithFormat:  
        NSString(@"No file exists at %@. Check the path and try  
        again.", @"informative text for NoSuchFile alert"),  
        [[self currentDiaryDocumentTextField] stringValue]]];  
    return alert;  
}  
  
- (NSAlert *)alertWrongFileType {  
    NSBeep();  
    NSAlert *alert = [[[NSAlert alloc] init] autorelease];  
    [alert setMessageText:NSString(@"The file is the wrong  
        type.", @"message text for alertWrongFileType alert")];  
  
    [alert setInformativeText:[NSString stringWithFormat:  
        NSString(@"The file at %@ is not a Vermont Recipes  
        Chef's Diary file or a Rich Text Format file. Enter a path  
        ending in ".vrdiary" or ".rtf" and try again.",  
        @"informative text for alertWrongFileType alert"),  
        [[self currentDiaryDocumentTextField] stringValue]]];  
    return alert;  
}
```

5. You have finished the Chef's Diary pane of the preferences window. Build and run the project to test it. To test the Chef's Diary Window section, perform

the same tests you performed at the end of Step 4, but using the diary window instead of the recipes window. To test the Printing section, perform the same tests you performed on the All Print Jobs section of the Print panel in Recipe 9.

To test the Autosaving section, simply choose a setting; then make some changes to the Chef's Diary and see how long it takes for an autosaved copy to appear.

To test the Document section, make sure that you have saved a Chef's Diary; then open the preferences window and go to the Chef's Diary pane. You should see the path to the document in the text field. Type a spurious letter into the field so that the path is invalid, and then press Return. You see an alert sheet explaining the error. Enter the path of an existing file that is neither a Vermont Recipes diary document nor an RTF file, and you see a different alert. In either case, the text field reverts to its prior content after you dismiss the alert. Try saving a good Chef's Diary document under a different name to make the new file the current Chef's Diary document. If you left the preferences window visible on the screen, you saw the path in the text field change the instant you dismissed the Save panel. Close the new Chef's Diary, and then type the path of the old Chef's Diary document into the field and press Return. Now when you choose File > Open Chef's Diary, the old Chef's Diary opens because you made it the current Chef's Diary by typing its path into the text field.

Step 6: Build and Run the Application

You tested the preferences window thoroughly after you completed each of the three panes. It is still worth testing them again now that you've finished this recipe, but I don't have any new tests to suggest. Try to put yourself in the shoes of a user and do everything wrong that you can think of. Hopefully, the application will successfully stand up to your abuse.

Step 7: Save and Archive the Project

Delete all the snapshots you've accumulated and quit the running application. Close the Xcode project window, discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like **Vermont Recipes 2.0.0 - Recipe 10.zip**. The working Vermont Recipes project folder remains in place, ready for Recipe 11.

Conclusion

You have completed a fairly complex preferences window for the Vermont Recipes application. You fulfilled promises made in earlier recipes to provide a user interface to enhance the flexibility of the application in specific areas, such as allowing the user to set another diary document or even an RTF file as the new current Chef's Diary. You also set up a user interface where the user could customize other features of the application that might reasonably be the subject of different user preferences, such as the standard size of the recipes window and the diary window.

My goal in this recipe was twofold. Most important, I wanted to show you basic techniques for implementing a preferences window. But I also wanted to demonstrate some techniques that aren't always used in application preferences, such as making sure that every change made by the user takes effect immediately in order to give the user immediate feedback. An application that feels responsive is a better application.

In Recipe 11, you learn how to provide a help book to guide the user through the features of the application.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 10.

Class Reference and Protocol Documents

NSUserDefaults Class Reference

General Documentation

User Defaults Programming Topics for Cocoa

Preferences Programming Topics for Core Foundation

Property List Programming Guide

Preference Pane Programming Guide

Technical Q&A QA1552: Enabling the application menu's "Preferences" menu item on Mac OS X

Sync Services Programming Guide (Syncing Preferences)

Add Apple Help

Only one menu item doesn't yet work as it should in the Vermont Recipes application: the Vermont Recipes Help menu item. The existing Vermont Recipes Help command, provided by the MainMenu.nib file in the Cocoa document-based application template, does work, but as is true of all Cocoa applications by default, it only presents an alert panel explaining, "Help isn't available for Vermont Recipes" (**Figure 11.1**).

You've probably seen similar alerts on applications undergoing beta testing, but this isn't acceptable for a finished application. At the very least, even the simplest application should open a read-me document from the Help menu, if only because it's so easy to take advantage of this opportunity to explain and promote your application. In Step 2 of Recipe 5, you did that by adding a Read Me menu item to the Help menu to open a short RTF document you created in TextEdit. Now you will do what every application should do: implement comprehensive HTML-based Apple Help that your users can read with Apple's HelpViewer application.

HIGHLIGHTS

Implementing a help book bundle for Snow Leopard

Implementing a help book for Leopard



FIGURE 11.1 What the Vermont Recipes Help command displays when you don't implement Apple Help.

This recipe is about the new help book structure that Apple made available to developers in Snow Leopard. The new structure existed before Snow Leopard, but it was available only for use by Apple's applications. In older versions of Mac OS X, localized versions of a third-party application's help book were scattered throughout the application package's Resources folder, each language's help book located in the appropriate language folder, such as English.lproj. In Snow Leopard, Apple recommends that all localized help books be gathered into a single help bundle in the application's Resources folder. The new help bundle has its own Resources folder,

and that Resources folder contains separate language folders holding the help files for that language. The help book you create in this recipe works when you run Vermont Recipes under Snow Leopard, but not when you run it under Leopard.

The current Apple documentation for Apple Help as I write this, *Apple Help Programming Guide*, was updated in May 2009 to document this new bundle structure. However, as the document recites at the beginning, it is written for Mac OS X 10.4 and newer. As a result, it has some internal inconsistencies. The most noticeable of these is the statement that the `AppleTitle` HTML meta tag is required. In fact, it is not required in a Snow Leopard help book structured as a bundle. This recipe lays out in detail how to set up a Snow Leopard help bundle. At the end, it explains how to set up a separate, old-style help book for use when the application runs under Leopard.

The least important difference between the Leopard and Snow Leopard versions of Apple Help is the odd fact that the HelpViewer application is named *Help Viewer* (two words) in Leopard and *HelpViewer* (one word) in Snow Leopard. I will attempt to honor this distinction by calling it Help Viewer when discussing Apple Help in Leopard.

Step 1: Implement an HTML-Based Apple Help Bundle for Snow Leopard

Apple Help has been Apple's preferred technology for delivering comprehensive online application help for a long time, going back well before Mac OS X. It is HTML based, with HTML 4.01 as its foundation.

Using HTML has many advantages, chief among them the fact that it is a mature, widely adopted, standards-based markup language supporting rich user interface design and presentation capabilities. It gives developers the ability to use a large variety of existing WYSIWYG HTML editors to design and write attractive help books. It also allows users to view application help in Web browsers other than Apple's own HelpViewer application, whether on the user's computer, on a network, or over the Internet. HTML's hypertext links are ideal in a help book, making it very easy for users to follow cross-references to related topics at any time. It can also ease the task of porting help to or from another platform, such as Windows.

Implementing Apple Help also has advantages within your application. For example, the Spotlight For Help search field at the top of the Help menu is provided automatically in Mac OS X 10.5 Leopard and newer, but by default it only searches your application's menus. If you implement Apple Help, the search field includes your

entire help book (**Figure 11.2**). If you are forced to use a cross-platform help system instead of Apple Help, you can include your help system in the Spotlight For Help search field using the `NSUserInterfaceItemSearching` protocol.



FIGURE 11.2 The Vermont Recipes Spotlight For Help search field with Apple Help.

Implementing Apple Help also lets you integrate your help book into other elements of your application—for example, by adding Help buttons to alerts and dialogs to take the user directly to the relevant section of your help book. The Apple Help API, available for Cocoa applications in the `NSHelpManager` class, makes it very easy to implement this feature.

Apple's `HelpViewer` application supports several features that are not part of the HTML standard, by using custom HTML tags that standard Web browsers simply ignore. In Mac OS X, one of the more important of these is AppleScript support. You can embed AppleScripts in your help book and provide clickable links enabling the user to perform a variety of automated tasks to illustrate the use of the application. Even if your application is not scriptable, your help book can include items such as a link to launch the application and a link to reveal it in the Finder. If your application is scriptable, you can provide links enabling your users to exercise specific application features while reading the help book. You can also launch movies and play sounds. Movies are a particularly intriguing possibility, since several utilities are available that record screen activity so that a user can then play back from your help book. Instead of *telling* users what to do, you can *show* them what to do.

Starting with Snow Leopard, the recommended location for all localizations of your help book files and folders is in a single bundle in the Resources folder in your application package. The use of multiple help books in an application is being deprecated. Gathering all of the help book's files and folders for all localizations into a bundle makes it easier to separate the creation and maintenance of your help book from the Xcode project, which may be especially convenient if you have a separate team writing your help book.

Whether you use the new unified Snow Leopard structure or the old structure that left localized help books scattered throughout the application's Resources folder, it is important to add the help files to the application package when it is built. To do

this, you turn on the “Create Folder References for any added folders” option when adding the help files to the built product, as described in a moment. Placing the help bundle in the application package ensures that it is installed automatically whenever a user installs your application by dragging and dropping it, in keeping with the Mac OS X philosophy of putting everything related to the application in one place.

In this step, you create a very simple help book and make it available from the Vermont Recipes application’s Help menu. You’ll leave out most of the text that would go into a real help book, focusing instead on the organization and mechanics of making the help book work.

These are the steps required to implement an Apple Help book:

- Design and write the HTML content using any HTML authoring application that supports the HTML 4.01 standard at <http://www.w3.org/TR/html4/>.
- Assemble and organize the files, including required auxiliary files such as a title page meeting the XHTML 1.0 standard at <http://www.w3.org/TR/xhtml1/>.
- Index the files using Apple’s Help Indexer utility.
- Register the help book in the application’s Info.plist file.

Here, you will take these steps out of the usual order, because this book is about creating a working application, not about writing effective help content. In this step, you spend only a moment to create minimal help content consisting of a title page containing an icon and the title *Vermont Recipes Help*. The rest of this step shows you how to make this minimal content actually show up in a HelpViewer window when the user chooses Help > Vermont Recipes Help. In Step 2, you will create a little more content, but only enough to give you the opportunity to add additional topic, task, and navigation pages and to learn a little more about the finer points of Apple Help.

1. Leave the archived Recipe 10 project folder in the zip file where it is, and open the working Vermont Recipes subfolder.

Increment the version in the Properties pane of the Vermont Recipes target’s information window from 10 to **11** so that the application’s version is displayed in the About window as *2.0.0 (11)*.

2. The design, organization, and content of the help book are entirely up to you, except for a small number of specific requirements that are spelled out in the *Apple Help Programming Guide*.

If you want to emulate the style of Apple’s Snow Leopard help books, you don’t have to reverse-engineer them. Instead, you can examine them and their HTML source in detail to see exactly how Apple does it. Since Apple no longer provides

a sample help book for developers, as it once did, Apple's help bundles are good places to find ideas and HTML source.

For general help books that aren't associated with a particular application, Apple places the help bundle in the Library folder in the Local domain. Apple reserves the Library/Documentation/Help folders in the Local domain and the User domain for its own use, so don't put your Help book there. Apple also reserves the existing nonapplication .help bundles in these locations, so you should not edit them to add your own help content.

For an example, open the MacHelp.help bundle on your computer at /Library/Documentation/Help. Control-click (or right-click) the MacHelp.help bundle to open a contextual menu, and then choose Show Package Contents. In the help bundle's window, navigate to Contents/Resources/English.lproj. There you find several files and folders, many with somewhat odd names (**Figure 11.3**). The names of the folders are derived from the long history and tradition of Apple Help, but you are not required to use them. The pgs folder contains separate HTML files for each page in the help book, and the xpgs folder contains a single HTML file for a human-readable index page in the help book. Apple names its HTML files using numbers or cryptic names, but I encourage you to use descriptive names for ease of maintenance. The scpt folder is especially interesting. It contains several compiled AppleScript scripts. The AppleScript text has not been stripped out of these scripts, and the *Apple Help Programming Guide* invites you to reuse them in your own help book.

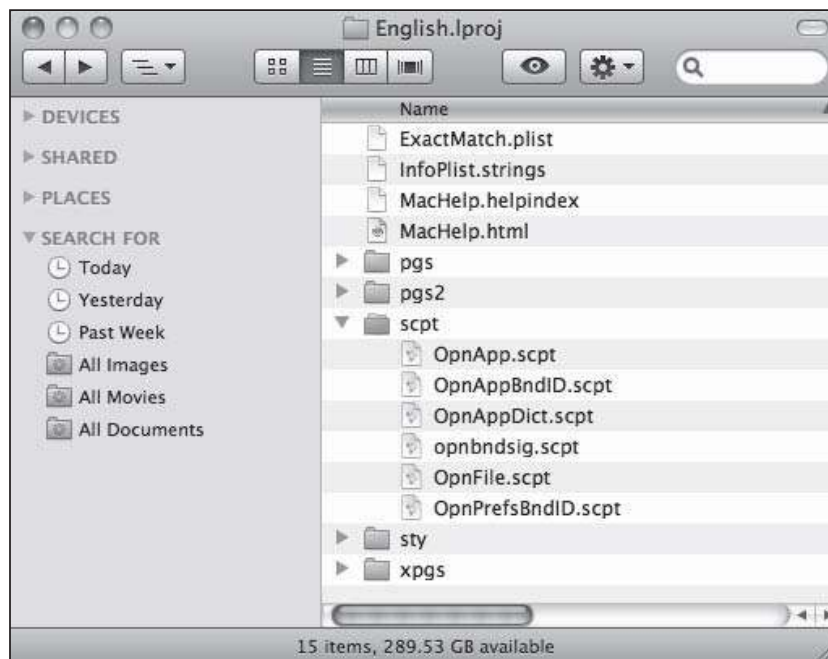


FIGURE 11.3
The English.lproj
folder of Apple's
MacHelp.help bundle.

The MacHelp.html file at the root level of the English.lproj folder is the help book's title page. You can double-click it to open it in Safari (**Figure 11.4**). In Safari, choose View > View Source to see its HTML source (**Figure 11.5**). Alternatively, drop the title page on any text editor that is configured to show the HTML source rather than displaying it as a Web page.



FIGURE 11.4
The MacHelp.html title page shown in Safari.

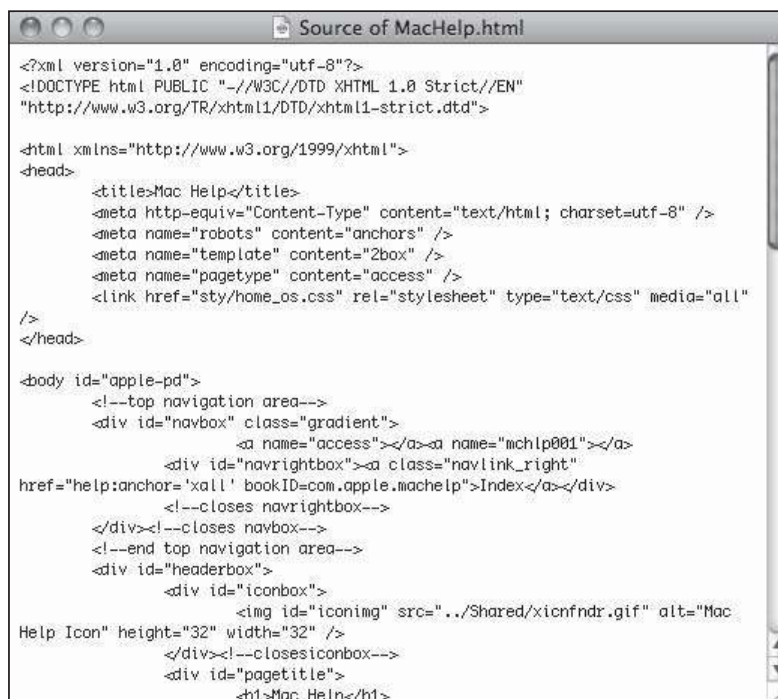


FIGURE 11.5
HTML source for the MacHelp.html title page.

The MacHelp.helpindex file in the English.lproj folder was generated using Apple's Help Indexer utility or the hiutil command-line tool on which Help Indexer is based. The HelpViewer application uses this index file for fast searching of the help

book's content. The `ExactMatch.plist` file was built by hand to help search for special terms used in the help book that might otherwise not be indexed or that might generate too many irrelevant hits. The `InfoPlist.strings` file contains the localized name of the help book.

Apple applications written for Snow Leopard place the help book bundle in the Resources folder of the application package, instead of a Library/Documentation/Help folder, just as you should do. Mail and iChat are examples. Their help bundles are organized the same way `MacHelp.help` is organized. The *Apple Help Programming Guide* encourages you to examine `Mail.help` to see how to write your own help book.

You can also examine help books in third-party applications, but few if any use the new Snow Leopard structure as I write this recipe.

Prior to Snow Leopard, you would typically arrange for an English-language help folder named Vermont Recipes Help to be placed in the `English.lproj` subfolder of the Resources folder of the built application package. Localized versions of the Vermont Recipes Help folder would be placed in separate language folders alongside the `English.lproj` folder, resulting in multiple localized help books scattered throughout the application package's Resources folder.

In Snow Leopard, the recommendations have changed. Now, the finished help folder should be organized according to the rules governing Mac OS X bundles. It contains its own Resources folder with its own `English.lproj` and other language subfolders containing localized versions of the help files. The Snow Leopard help bundle for Vermont Recipes is named `VermontRecipesHelp.help`, and it is placed at the top level of the Resources folder of the built application package.

The new arrangement makes it easier to segregate help book production from application programming. Instead of hunting down localized help folders one by one in all of the separate language folders in the project folder, you only have to find the one help bundle, which you typically store separately from the project folder. The new arrangement also allows resources that don't require localization, such as images, to be shared by all localizations, in a shared folder at the root level of the help book bundle's Resources folder.

To begin creating your help book for Vermont Recipes, create a new folder anywhere using the Finder. Place it in your Vermont Recipes 2.0.0 folder or in your Documents folder, for example, but don't place it in the Vermont Recipes project folder. This is going to be a Snow Leopard help bundle, so name it *VermontRecipesHelp.help* with the `.help` file extension. Open it, and create in it a subfolder and name it *Contents*. Open the Contents subfolder, and in it create a *Resources* subfolder. Open the Resources subfolder, and in it create an *English.lproj* subfolder. From now on, you will add content files, some special files, and subfolders to this subfolder in the `VermontRecipesHelp.help` bundle.

The final hierarchy is `VermontRecipesHelp.help/Contents/Resources/English.lproj/` (**Figure 11.6**). You will add the bundle with all of its new subfolders to the project shortly.

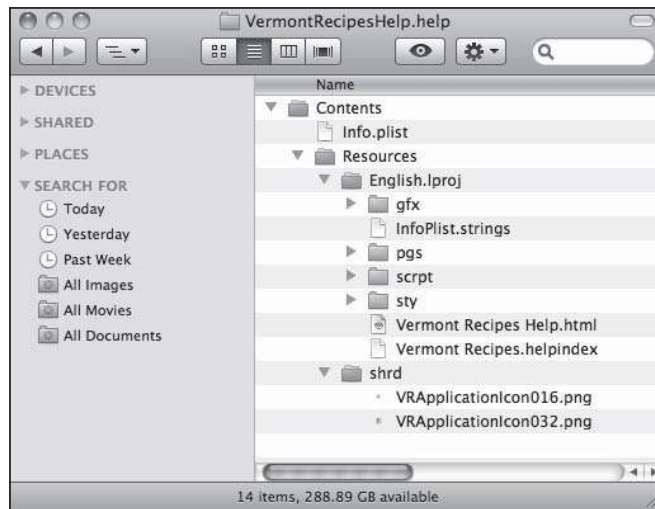


FIGURE 11.6 The VermontRecipesHelp.help bundle hierarchy after adding content.

3. Start populating the `English.lproj` subfolder with files and subfolders now. Using the Finder, create subfolders named `gfx` (for *graphics*), `pgs` (for *pages*), `sty` (for *style sheets*), and `scrpt` (for *scripts*). These will hold, respectively, localized image files, individual help book pages other than the title page, CSS style sheets, and compiled AppleScript scripts. The naming of these files conforms to Apple's instructions in the *Apple Help Programming Guide*. There is nothing magic about the names, and even Apple thinks different, calling its scripts folder *scpt* instead of *scrpt*.

In addition, create a subfolder in the `Resources` folder alongside the `English.lproj` subfolder, and name it *Shared*. This subfolder will hold files to be shared by all localizations of the help book, primarily graphics files that do not require localization. The *Apple Help Programming Guide* suggests calling it the *shrd* folder, but it looks a little odd in a long list of language folders in the `Resources` folder. Apple's Snow Leopard applications name it the *Shared* folder.

Drag the `VRApplicationIcon016.png` and `VRApplicationIcon032.png` files from wherever you saved them in Recipe 8 into the *Shared* subfolder in the `Resources` folder you just created. The 16 by 16 icon will be used in the Vermont Recipes Help menu item in HelpViewer's Library menu. The 32 by 32 icon is the size typically used on the title page of help books. In Step 2, you will experiment with somewhat larger icon sizes, so drag the `VRApplicationIcon048.png` and `VRApplicationIcon064.png` files into the *Shared* subfolder as well.

4. Next, create the title page of your help book. To demonstrate that you don't need fancy HTML authoring tools to create a help book, use TextEdit and write raw HTML. To use TextEdit to write raw HTML, you should set TextEdit's Open and Save preferences in the "When Opening a File" section to "Ignore rich text

commands in HTML files,” and in the HTML Saving Options section to HTML 4.01 Strict, Embedded CSS, and Unicode (UTF-8). When creating a help book title page, set the “Document type” setting to XHTML 1.0 Strict.

The suggestion to use XHTML 1.0 for the title page has to do with execution speed. It is not an absolute requirement. In fact, Apple advises against using the XHTML 1.0 standard in Snow Leopard applications that support very old versions of Mac OS X.

In TextEdit, create a new file and save it in the English.lproj subfolder in your VermontRecipesHelp.help bundle using UTF-8 encoding. Name it *Vermont Recipes Help.html*. This is the help book’s title page.

5. Enter the following HTML in the empty Vermont Recipes Help.html file. For this step, keep it really simple and include only enough content to show what page you’re looking at, so that you can quickly build it into the application and test it.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
  <title>Vermont Recipes Help</title>
  <meta name="AppleTitle" content="Vermont Recipes Help.html"/>
  <meta name="AppleIcon" content="../Shared/VRApplicationIcon016.png"/>
  <meta name="robots" content="anchors"/>
  <link href="sty/access.css" rel="stylesheet" media="all"/>
</head>
<body>
  <h3>
    Vermont Recipes Help</h3>
</body>
</html>
```

This is not a tutorial on HTML, so I won’t explain what all these HTML tags do, nor will I offer many suggestions regarding other HTML tags you could add. If you’re going to write your own help book, you must understand at least the basics of HTML. There are many books available. Otherwise, hire somebody to write the help book for you.

The first three lines in the title page are required above the `<head>` tag in all XML files, as explained in the *Apple Help Programming Guide*.

Set the standard HTML `title` element under the `<head>` tag to Vermont Recipes Help. Other applications such as Web browsers use it to display the title of the page.

Two of the `meta` tags you see here—for the names `AppleTitle` and `AppleIcon`—are custom Apple Help tags. They are legacy tags, for help books that run under Mac OS X 10.5 Leopard and older. Apple does not include them in its Snow Leopard–only applications, because the same information is provided in the `HPDBookAccessPath` and `HPDBookIconPath` entries in the `Info.plist` file in the new help bundle structure, which you will encounter in a moment. The *Apple Help Programming Guide* erroneously states, without qualification, that `AppleTitle` is required. However, neither it nor `AppleIcon` is listed in the table referenced in the *Guide* as a complete list of recognized `meta` tag names for Snow Leopard. You include them here only because you will use this title page in the Leopard version of the help book at the end of this recipe. They are ignored by Snow Leopard.

The name in the `AppleTitle` `meta` tag must be identical to the name of the title page of the help book, with or without the `.html` file extension. It is actually the path to the title page, but since it is recited in the title page itself and is at the same level of the hierarchy as the title page, it doesn't need any path separators. Help Viewer under Leopard and older uses this setting to identify the page as the help book's title page, which Help Viewer displays when the user chooses Help > Vermont Recipes Help. The title page is sometimes referred to as the default page, the landing page, the start page, or the access page.

The `AppleIcon` tag refers to the 16-by-16-pixel PNG application icon file you created in Step 11 of Recipe 8. Help Viewer's Library menu (which doubles as the Home button) displays this icon in the Vermont Recipes Help menu item when a user wants to open Vermont Recipes Help directly from Help Viewer without first opening it from the Vermont Recipes Help menu.

The `AppleIcon` tag and the `img` element in the body define the path to the 16 by 16 and 32 by 32 application icons that appear in the Library menu and on the help book's title page as `../Shared/VRApplicationIcon016.png` and `../Shared/VRApplicationIcon032.png`, respectively. The title page is in the `English.lproj` folder, so you have to move up one level to find the `Shared` subfolder of the `English.lproj` folder where the shared application icons are located. Note that the corresponding `HPDBookIconPath` entry in the `Info.plist` file for Snow Leopard does not include the leading `../`; the *Apple Help Programming Guide* states that it is the path relative to the help bundle's `Resources` folder.

The last two elements under the `head` tag are required only in the title page, according to the *Apple Help Programming Guide*. This isn't accurate. First, the exact form of the `link` element depends on how you name the style sheets in the help book. You will change the `link` element in the title page in Step 2. In addition, every page that uses a style sheet also requires a `link` element. Apple's applications use different style sheets for different types of pages.

6. A Snow Leopard help bundle needs an Info.plist file of its own. The *Apple Help Programming Guide* lists what it suggests are the required keys, although some of them are actually optional. An easy way to create the Info.plist file is to use Apple's Property List Editor application, in your Developer folder at Applications/Utilities. Examine the Info.plist file for the Snow Leopard help bundle in the downloadable files for Recipe 11. It's in a separate folder of its own.

The CFBundleSignature must be `hbwr` in all help files, according to the *Apple Help Programming Guide*.

The paths in the `HPDBookAccessPath` and `HPDBookIndexPath` entries start from the English.lproj folder. The `HPDBookAccessPath` entry is the path to the title page, which is sometimes called the *access page*. I have included the .html file extension here and in the `AppleTitle` meta element in the title page to emphasize that this is not a reference to the help book itself but only the path to its title page. Some published articles get this wrong. The .html file extension is left out in most third-party applications, and most applications name the title page the same as the help book itself, which accounts for the confusion. Apple's Snow Leopard applications include the .html file extension in the `HPDBookAccessPath` entry. The path in the `HPDBookIconPath` entry starts from the Resources folder, and it therefore does not need the leading `../` that is required in the `AppleIcon` meta tag in the title page. The `HPDBookIndexPath` entry is discussed in greater detail in a moment, in connection with the use of the Help Indexer utility.

I have included all of the valid `HPDBook...` entries in the list, but the list gives most of them no values. Apple's documentation offers no explanation as to what these entries do, and Apple's Snow Leopard applications leave most of them out.

Save the Info.plist file at the root level of the help bundle's Contents folder.

7. The `HPDBookTitle` entry must be localized in an InfoPlist.strings file. Create the file using TextEdit or any other application that can create plain text files. Enter the following as its content:

```
/* Localized versions of Info.plist keys */
HPDBookTitle = "Vermont Recipes Help";
```

Save the InfoPlist.strings file at the root level of the English.lproj subfolder in the help bundle's Resources subfolder. Be sure to save it using UTF-16 encoding, as required for all .strings files. In TextEdit, you can do this by setting the Encoding preference to Unicode (UTF-16) in the HTML Saving Options section of the "Open and Save" pane in TextEdit's preferences window.

8. There are several additional steps you can take to refine the effectiveness of the index for the help book, such as adding keywords and abstracts, but leave those until you have finished writing the help book's content.

Index the help book using Apple's Help Indexer utility, in the Applications/Utilities folder of your Developer folder. The default settings are fine for an English help book. To use the utility, drag the English.lproj subfolder onto the Help Indexer icon and click the Create Index button; then quit Help Indexer. The index file is saved under the name *English.lproj.helpindex* in the English.lproj subfolder, alongside the Vermont Recipes Help.html title page. This is the right place, but the name is unfortunate. Manually change the name from *English.lproj.helpindex* to **Vermont Recipes.helpindex** in the Finder. This matches the value of the `HPDBookIndexPath` entry in the Info.plist file you just created. You should have been able to edit the path of the index file in the Help Folder text field in the Help Indexer's window, but due to an apparent bug, the text field does not allow you to scroll the long path string so as to edit the filename.

Be sure to re-index the help book every time you make any changes to its content.

The Help Indexer is based on the `hiutil` tool in Snow Leopard. You can read technical information about it in the `hiutil(1)` man page.

9. Now go to the Xcode project window and select the Resources group in the Groups & Files pane on the left. From the action menu in the project window's toolbar, choose Add > Existing Files, or from the menu bar choose Project > Add to Project. Then navigate to the VermontRecipesHelp.help bundle you just created, select it, and click Add to add it to the project.

In the next sheet, make very sure to select the "Create Folder References for any added folders" radio button before clicking the Add button, to force Project Builder to reference only the top level of the VermontRecipesHelp.help bundle, wherever you saved it, without copying its contents into the project folder. When you build the project, Xcode will find the help bundle and copy it into the built application package. If you move the help bundle later, you will have to re-add it to the project before building it again. If you leave it in the same place where you created it, you will not have to add it to the project again even if you add additional subfolders and files.

Until now, I have always advised you to make sure that both the Vermont Recipes and Vermont Recipes SL targets are selected before you click Add. For this recipe, however, you should select only the Recipes SL target, leaving the Vermont Recipes target deselected. The help book you are adding will work only when the application is running under Snow Leopard. In Step 8, you will set up a legacy-style help system for the Leopard target.

10. In the Groups & Files pane of the main project window, drag the Vermont-RecipesHelp.help group into the Resources group, if necessary. This shouldn't be necessary if the Resources group was already selected when you added the help bundle.

11. Next, you must register the help book. In Cocoa, you aren't supposed to have to write any code to do this. In fact, however, you do have to write some code to support the display of help abstracts in search results, as explained in Step 5. For now, however, register the help book for most purposes by simply adding entries to the application's Info.plist file.

In Xcode, open the `Vermont_Recipes-Info.plist` file and add two entries. One new entry is `CFBundleHelpBookFolder`, which should be set to **VermontRecipes-Help.help**, the title of the help book bundle. The other new entry is `CFBundleHelpBookName`, which you should set to **com.quecheesoftware.vermontrecipes.help**, the `CFBundleIdentifier` that you set for the help book bundle in its Info.plist file. Prior to Snow Leopard, the help book folder was typically a human-readable name like *Vermont Recipes Help*, and the help book name was also a human-readable name, typically identical to the help book name, such as *Vermont Recipes Help*. Starting with Snow Leopard, if you use the help bundle format, the help book folder must end in `.help`, and the help book name must be a bundle identifier that you do not localize.

12. In the MainMenu nib file, the Vermont Recipes Help menu item in the Help menu must be connected to the First Responder proxy for the application object's `showHelp:` action method. The Cocoa document-based application template already connected this for you, so you don't need to worry about it now.
13. Build and run the application and choose Help > Vermont Recipes Help. Apple's HelpViewer launches and your application's help book appears. Think how much money you will save on printing costs when you begin to distribute your finished application.

There is one other feature that is now working: The Vermont Recipes help book appears in HelpViewer's Library menu. HelpViewer's Library menu is the pop-up button in the toolbar with the image of a little house. Open it, and you see a long menu of help books for all the applications on your computer that include a help book. Vermont Recipes Help is included in the menu along with its application icon. Users will be able to open your help book even without running the Vermont Recipes application.

Step 2: Add Topic, Task, and Navigation Pages

A help book is useless without content. In this step, you complete the title page, which is a form of navigation page, and you add several topic pages. Topic pages and task pages require very similar HTML source. You complete only one of the topic pages in this step, About Vermont Recipes, to illustrate the process.

The *Apple Help Programming Guide* describes each type of page. In summary, a topic page, also referred to as an overview page, describes a concept or subject of general importance in the application; a task page lists steps to follow to carry out a particular operation; and a navigation page acts as a table of contents to let you move from topic to topic in an organized fashion. In relatively simple applications, it is desirable to limit the use of navigation pages and instead to provide navigation by using links within topic pages. In more complicated applications, it is preferable to use pages that list subtopics to break large subjects into smaller pieces.

1. Start by fleshing out the existing title page, Vermont Recipes Help.html. It is a navigation page, acting as the help book's table of contents. Follow the model of the Mail help book's title page. It breaks the title page into several sections and subsections using HTML `div` elements. Using the `id` attribute, it calls them the `headerbox` and `columnshell` sections. The `headerbox` section holds the application icon and the title of the help book, each in its own subsection named, respectively, `iconbox` and `pagetitle`. The `columnshell` section is divided into `leftcolumn` and `rightcolumn` subsections. Each section or subsection contains content that is formatted using a class defined in a CSS style sheet in the `sty` subfolder. By providing style sheets in the Vermont Recipes help book that are like Apple's style sheets, you can achieve the same appearance as Apple's help books for a consistent user experience.

Replace the HTML source within the body element in the Vermont Recipes Help.html title page you created in Step 1 with the following:

```
<div id="headerbox">
  <div id="iconbox">
    
  </div>
  <div id="pagetitle">
    <h1>Vermont Recipes Help</h1>
  </div>
</div>
<div id="columnshell">
  <div id="leftcolumn">
    <div class="leftcolpadding">
      <p class="subheadleft"><a href="pgs/AboutVermontRecipes.html">
        About Vermont Recipes</a></p>
      <p class="leftcolumnntext">Look up recipes and ingredients,
        create shopping lists, keep a Chef's Diary.</p>
      <p class="subheadleft"><a href="pgs/Recipes.html">
        Recipes</a></p>
    </div>
  </div>
</div>
```

```

        <p class="leftcolumnntext">Review ingredients and utensils,
            follow instructions to cook a delicious dish.</p>
        <p class="subheadleft"><a href="pgs/ShoppingLists.html">
            Shopping lists</a></p>
        <p class="leftcolumnntext">Create and print shopping
            lists.</p>
        <p class="subheadleft"><a href="pgs/ChefsDiary.html">
            Chef's Diary</a></p>
        <p class="leftcolumnntext">Keep a Chef's Diary.</p>
    </div>
</div>
<div id="rightcolumn">
    <div class="rightcolpadding">
        <div id="contentframe">
            <p class="headboxtopright">FEATURED TOPICS</p>
            <p class="boxlink"><a href="pgs/CreatingRecipes.html">
                Creating recipes</a></p>
            <p class="boxlink"><a href="pgs/ListingIngredients.html">
                Listing ingredients</a></p>
            <div id="rule">
                <hr/>
            </div>
            <p><a href="http://www.quecheesoftware.com"
                class="boxlinkapple">www.quecheesoftware.com</a></p>
        </div>
    </div>
</div>
</div>

```

Next, copy the `home_os.css` file from the Mail.app application package at `Contents/Resources/Mail.help/Contents/Resources/English.lproj/sty` into the working `VermontRecipesHelp.help` folder at `Contents/Resources/English.lproj/sty`. Also, change the title page's link to this style sheet in the last line after the `<head>` tag to `<link href="sty/home_os.css" rel="stylesheet" type="text/css" media="all"/>`, so that HelpViewer will use it when you open Vermont Recipes Help. The `home_os.css` file contains all of the styles that are referenced in the `id` attributes of the `<div>` tags in the title page, including the `apple-pd` style that specifies `font-family: 'Lucida Grande', Arial, sans-serif;`.

To see the results of the change, you may have to discard the help cache files so that new ones can be generated. Otherwise, the old help content will continue to appear in HelpViewer. To do this in Snow Leopard, open the `~/Library/`

Caches folder and drag the com.apple.helpd folder to the Trash. Do this before you build and run the application. You'll likely have to do this every time you make a change to the help book's content. You may also have to clean the project, and don't forget to re-index the help book's English.lproj folder.

When you're done, build and run the application, and choose Help > Vermont Recipes Help. You see a very Mac-like title page for your help book (**Figure 11.7**).



FIGURE 11.7 The finished Vermont Recipes Help title page.

2. To my eye, the application icon on the title page is a little too small. In Recipe 8, when you created the application icons, you created a couple of odd sizes in case you needed them for the help book or your application's Web site. To try one of the larger icon sizes in the help book, change the `iconbox` `div` element you just wrote in `Vermont Recipes Help.html` so that it contains this statement:

```

```

Ah yes, the 64 by 64 icon is much nicer.

3. Next, add some topic pages to the help book. In the title page, you linked to each of them using an HTML anchor link element to the topic page in the `pgs` folder.

As you did with the title page in Step 1, use TextEdit to create a new file. Save it in the `pgs` subfolder of the `English.lproj` subfolder in your `VermontRecipesHelp` help bundle using UTF-8 encoding. Name it *AboutVermontRecipes.html*.

Add the following HTML source to the topic page. This is the basic template you will use for each topic and task page, changing the `title` element and the `pagetitle` `div` element of each to the name of the page. You'll add content in a moment.

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>About Vermont Recipes</title>
    <link href="../../sty/task.css" rel="stylesheet" type="text/css"
        media="all"/>
</head>

<body id="apple-pd">
    <div id="headerbox">
        <div id="iconbox">
            
        </div>
        <div id="pagetitle">
            <h1>About Vermont Recipes</h1>
        </div>
    </div>
</body>

</html>

```

Create each of the remaining topic pages linked in the title page: `ChefsDiary.html`, `CreatingRecipes.html`, `ListingIngredients.html`, `Recipes.html`, and `ShoppingLists.html`. Be sure to change the `title` element and the `pagetitle` `div` element of each.

I have not found any documentation specifying capitalization rules for help page titles, but a quick review of Mac Help and help books for Apple's applications makes clear that you should use sentence case. You should therefore enter the titles of the remaining topic pages as follows: **Chef's Diary**, **Creating recipes**, **Listing ingredients**, **Recipes**, and **Shopping lists**.

In all of the topic pages, you have set the `link` element after the `head` tag to refer to a `task.css` style sheet. Copy it now from the Mail.app application package at `Contents/Resources/Mail.help/Contents/Resources/English.lproj/sty` into the working `VermontRecipesHelp.help` folder at `Contents/Resources/English.lproj/sty`. It contains some of the same styles as the `home_os.css` style sheet you used in Step 1, such as `apple-pd`, but it adds some new styles appropriate to a topics or tasks page.

Now return to the AboutVermontRecipes.html topic page and add some content. Insert the following HTML source between the <body> and </body> tags in the AboutVermontRecipes.html topic page you just created:

```
<div id="headerbox">
  <div id="iconbox">
    
  </div>
  <div id="pagetitle">
    <h1>About Vermont Recipes</h1>
  </div>
</div>
<div id="introbox">
  <div class="introprimtext">
    <p class="introprimtextlongtext">
      With Vermont Recipes, you can collect an unlimited number of recipes,
      then call one up using powerful search tools whenever you get the urge
      to cook up a special dish. Decide how many guests to invite, then
      print a shopping list tailored to the correct quantities. When you're
      ready, open the recipe on your Mac beside the stove and get started.</p>
    </div>
    <div class="introprimtext">
      <p class="introprimtextlongtext">
        Vermont Recipes also provides a Chef's Diary, to help you capture and
        organize your most memorable culinary experiences. The Chef's Diary
        includes:</p>
      <div id="introauxlist">
        <div class="introauxoption">
          <div class="introauxbullet"></div>
          <div class="introauxcontent">
            <ul id="introauxoptionbulletlongtext"><li>
              <p class="listparagraph">
                An entry title with the current date and time, so you'll never forget
                when this happened.</p></li>
            </ul>
          </div>
        </div>
      </div>
      <div class="introauxoption">
        <div class="introauxbullet"></div>
        <div class="introauxcontent">
```



```

        <ul id="introauxoptionbulletlongtext">
            <li>
                <p class="listparagraph">
                    <p class="listparagraph">
Tag lists, so you can tag your experiences to find them easily
later.</p></p></li>
            </ul>
        </div>
    </div>
</div>
</div>
<div id="rule">
    <hr/>
</div>
<div id="linkinternalbox">
    <h3>Related Topics</h3>
    <p class="linkinternal"><a href="../pgs/Recipes.html">
        Recipes <span class="linkarrow"></span></a></p>
    <p class="linkinternal"><a href="../pgs/ChefsDiary.html">
        Chef's Diary <span class="linkarrow"></span></a></p>
</div>
</div>

```

4. Before building and running the application, move the com.apple.helpd file to the Trash, re-index the help book, and clean the project. Then build and run the application, choose Help > Vermont Recipes Help, and click About Vermont Recipes in the title page. You see a fully formed topic page, complete with a Related Topics box (**Figure 11.8**).

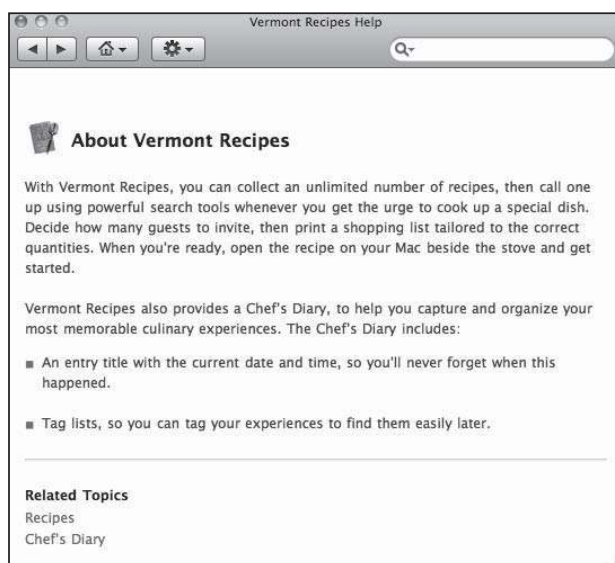


FIGURE 11.8
The finished About Vermont Recipes topic page.

Click either of the Related Topics links, and you are immediately taken to the linked page.

Step 3: Add an AppleScript Link to a Topic Page

One of the more powerful features of Apple Help is its ability to run AppleScript scripts from clickable links. Since AppleScript has extensive capabilities to control the system, the Finder, and other applications, your help book can enable the user to carry out tasks described on a help page simply by clicking a link. A common way to use this feature is to let the user launch an application or open a file that is described in the help book.

In this step, you add to the About Vermont Recipes page a clickable link reading “Reveal Vermont Recipes in the Finder.” Apple’s MacHelp.help bundle contains several compiled scripts, one of which—OpnAppBndID.scpt—is able to open or reveal any application based on its bundle identifier. You add a copy of that script to the Vermont Recipes Help book and link to it on the About Vermont Recipes page.

1. Start by adding a copy of the OpnAppBndID.scpt file to the Vermont Recipes help bundle. The *Apple Help Programming Guide* advises you not to link to the file in the MacHelp.help help bundle, but it grants you permission to make a copy of it and use it in your application. Place it in the script subfolder of the English.lproj folder in the Vermont Recipes help bundle’s Resources folder.
2. To enable the user to run the script, add this HTML statement to the AboutVermontRecipes.html page, just before the `rule div` element near the end of the file:

```
<p><a href="x-help-script://com.quecheesoftware.vermontrecipes.help/
    script/OpnAppBndID.scpt?reveal,com.quecheesoftware.Vermont-Recipes">
    Reveal Vermont Recipes in the Finder
    <span class="linkarrow"></span></a></p>
```

If you’re conversant with AppleScript, open the script in the AppleScript Editor by double-clicking it. You’ll see that it takes a parameter, `completeParam`, which contains two text items, `actionToTake` and `appBndID`. The first should be passed in either as `open`, `ASDict`, or `reveal`. Here, you used `reveal`, which causes the script to reveal the designated file in the Finder. The second parameter takes the bundle id of the target application file, in this case `com.quecheesoftware.Vermont-Recipes`.

3. Save the AboutVermontRecipes.html file. Then move the com.apple.helpd file to the Trash, re-index the help book, and clean the project. Build and run the application, choose Help > Vermont Recipes Help, and click About Vermont

Recipes in the title page. In the About Vermont Recipes page, click the “Reveal Vermont Recipes in the Finder” link. After a brief pause, the folder containing the Vermont Recipes application file opens and the file is selected. For me, it is revealed in the Debug folder of the build folder in the project folder. For a user of the finished application, it should be revealed in the Applications folder.

You put the `OpnAppBndID.scpt` script in the script subfolder in the `English.lproj` localization of the help book. This is appropriate, because the script can, under certain conditions, display error messages that are in the English language. If you add other localizations to Vermont Recipes, you should revise a copy of the script so that it displays the error messages in the appropriate language, and then place it in the script subfolder of that language folder in the help book.

You should be aware that many scripts do not display messages, and they therefore do not require localization. Snow Leopard has added the ability to share nonlocalized files among multiple language folders, and you can place such scripts in the Shared folder you created in Step 1. To run `OpnAppBndID.scpt` as a shared script after placing it in the Shared subfolder of the Resources folder, for example, you would write the HTML anchor element like this:

```
<p><a href="x-help-script://com.quecheesoftware.vermontrecipes.help/
    Shared/OpnAppBndID.scpt?reveal,com.quecheesoftware.Vermont-Recipes">
    Reveal Vermont Recipes in the Finder
    <span class="linkarrow"></span></a></p>
```

Neither the script folder reference nor the Shared folder reference in the two versions of the HTML statement requires leading “../” path components. I assume that the path is relative to the Resources folder, and that HelpViewer uses the user’s language preference to search the correct language folder when it discovers that a shared folder is not being used.

Step 4: Use the HelpViewer *help:* Protocol

The HelpViewer application in Snow Leopard implements a `help:` protocol, which is a private protocol similar to the well-known `http:`, `file:`, `ftp:`, and `mailto:` protocols. This is different from the `x-help-script:` protocol you used in Step 3, where you created a link that runs an AppleScript script. There are several other ways you can use the `help:` protocol: to search for a specified term, to link to an anchor location, to generate a list from anchors, and to open help books in other applications. You will implement some of these in this step.

1. Start by implementing a `help:search` URL. Add this HTML source as the first statement before the `</body>` tag in the Recipes.html file:

```
<a href="help:search='recipes'  
    bookID=com.quecheesoftware.vermontrecipes.help">  
    Find all references to "recipes" <span class="linkarrow">  
    </span></a>
```

Save the Recipes.html file. Then move the `com.apple.helpd` file to the Trash, re-index the help book, and clean the project. Build and run the application, choose `Help > Vermont Recipes Help`, and click Recipes in the title page. You see the Recipes page with the new search link (**Figure 11.9**). Click the “Find all references to ‘recipes’” link. A new page is generated in HelpViewer listing three Vermont Recipes help book pages under the heading Help Topics and a topic under the heading Support Articles (**Figure 11.10**). Each of the Help Topics is a clickable link that takes you to the indicated page.



FIGURE 11.9 The Recipes page with a search link.



FIGURE 11.10 The results page generated by clicking the search link.

2. Implement a `help:anchor` URL next.

At the end of Step 2, you created a Related Topics section at the end of the About Vermont Recipes page, with links to the Recipes page and the Chef’s Diary page. You implemented those links using a hard-coded path to each

page file, such as `<a href= "../pgs/ChefsDiary.html">Chef's Diary</a>`. Hard-coded paths are inherently fragile. This link, for example, will break if you move the `ChefsDiary.html` file into a subfolder in the help bundle.

A `help:anchor` URL lets you specify a unique anchor for the target page, and it finds that anchor wherever it may be. This gives you the freedom to reorganize your help book at will, without disturbing cross-references.

Start by setting the anchors using standard HTML. In `Recipes.html`, add this statement immediately after the `<body>` tag:

```
<a name="recipespage"></a>
```

The anchor for the body of the Recipes page is now `recipespage`.

Insert a similar anchor right after the `<body>` tag in `ChefsDiary.html`, like this:

```
<a name="chefsdiarypage"></a>
```

Now open `AboutVermontRecipes.html` and edit the hard-coded links at the end. Replace `<a href= "../pgs/Recipes.html">Recipes` with this:

```
<a href="help:anchor=recipespage
      bookID=com.quecheesoftware.vermontrecipes.help">Recipes
```

Also replace `<a href= "../pgs/ChefsDiary.html">Chef's Diary` with this:

```
<a href="help:anchor=chefsdiarypage
      bookID=com.quecheesoftware.vermontrecipes.help">Chef's Diary
```

Indexing anchors requires you to turn on the “Index anchor information in all files” setting in the Help Indexer. It is turned off by default. In the Help Indexer window, click `Show Details` and select the checkbox.

Save all three files. Then move the `com.apple.helpd` file to the Trash, re-index the help book, and clean the project. Build and run the application, choose `Help > Vermont Recipes Help`, and click `About Vermont Recipes` in the title page. At the bottom, in the `Related Topics` section, click the links to `Recipes` and to `Chef's Diary`, and you are immediately taken to those pages, just as you were before. This time, however, the HTML source is more robust.

To make it easier to create cross-references to every page in your help book, you should get in the habit of creating an anchor for each as soon as you create it. Go back to all of your other topic pages now and add anchor tags to them, right after the `<body>` tag.

In addition, you really should go back to the title page, `Vermont Recipes Help`, `help`, and change all of the hard-coded links to anchor links using `help:anchor` URLs. I'll leave that to you.

3. Now use the `help:openbook` URL to open another help book. For this example, create a new topic page in anticipation of adding AppleScript support to Vermont Recipes, which you will do in Recipe 12. Create the new topic page using the same simple HTML source you used when you created `ShoppingList.html`. Simply duplicate `ShoppingList.html` in the Finder, rename it *AppleScriptSupport.html*, and change its internal references from “Shopping lists” to “AppleScript support.”

To see the result, add a link to the new AppleScript support page in the Featured Topics section of the Vermont Recipes Help title page. In `Vermont Recipes Help.html`, add this statement near the end, after the links to the two other featured topics:

```
<p class="boxlink"><a href="pgs/AppleScriptSupport.html">
    AppleScript support</a></p>
```

Now add this statement at the end of the existing statements following the `<body>` tag in the `AppleScriptSupport.html` file:

```
<p><a href="help:openbook=com.apple.ScriptEditor.help">
    Open AppleScript Editor Help</a></p>
```

Save the `AppleScriptSupport.html` and `Vermont Recipes Help.html` files. Then move the `com.apple.helpd` file to the Trash, re-index the help book, and clean the project. Build and run the application, choose `Help > Vermont Recipes Help`, and click `AppleScript support` in the Featured Topics section. In the AppleScript support page, click the `Open AppleScript Editor Help` link. After a short pause, the title page of the AppleScript Editor help book appears.

I have not discussed the `help:topic_list` URL, which generates a list from anchors scattered throughout a help book. This feature is for a more complex help book. When you get to the point where you need it, you will find it useful to generate a complete index of your help book. Read the *Apple Help Programming Guide* for instructions.

Step 5: Add Keywords and Abstracts

There are a couple of techniques you can use to enhance the user’s search experience. One improves the quality of the search, while the other makes it easier for the user to make sense of the search results.

1. You should add keywords to every page to ensure that a search finds the page even when an appropriate search word does not appear in the page content. You don’t want to write help content with one eye on the thesaurus to make sure you use every word that a user might think to include in a search. Instead, write

the content for understandability, and separately add keywords to ensure good search coverage.

I find it useful to add keywords before writing the content. Thinking about how a user might search for the topic at hand is a good warm-up exercise. It helps me to understand the scope of the issue and to write better content. If I end up with some words both in the content and the keyword list, no harm is done.

Apple even recommends including common misspellings in the keyword list. I find this a daunting challenge, because I have no idea how to anticipate the misspellings somebody might use.

While thinking up good keywords can be difficult, adding them to a help page is simple. Open the new `AppleScriptSupport.html` file, for example, and add this statement following the `<head>` tag:

```
<meta name="keywords" content="AppleScript, AppleScriptable, script,
    scripting, scriptable, scriptability, automate, automated,
    automation, automatic, automator, workflow, interapplication
    communication, message, messaging, OSA, Open Scripting Architecture,
    command, control, remote, Apple event, AE, Script Editor,
    AppleScript Editor">
```

I chose AppleScript for this example because I know a lot about the subject and can easily come up with a lot of relevant terms. Even I found some more, however, by a quick read through the Wikipedia article about AppleScript and browsing the books in my AppleScript library.

Adding keywords to the other pages is left as an exercise for the reader.

Save the `AppleScriptSupport.html` files. Then move the `com.apple.helpd` file to the Trash, re-index the help book, and clean the project. Re-indexing is especially important for this task, because it is only the indexing process that makes the keywords available to the search engine. Then build the application.

Instead of running the application right off, go to the Finder and choose `Help > Mac Help`. In `HelpViewer`'s search field, pull down the search menu and choose `Search All Books`. Then type one of the keywords, **OSA**, and press `Return`. The search results list on my computer yields 8 Help Topics and 7 Support Articles. The Help Topics include, at the bottom, an item labeled "AppleScript support" with the Vermont Recipes application icon beside it. I know this is a result of the keywords I added to the AppleScript support page, because I haven't yet used the term OSA anywhere else in the Vermont Recipes help book.

Now search all books for the term *AppleScript*. I get 15 Help Topics and 7 Support Articles on my computer. One of them is "AppleScript support," but it doesn't

belong to Vermont Recipes. Although this isn't documented, HelpViewer's search is limited to 15 hits. Vermont Recipes didn't make the cut.

Build and run the application, choose Help > Vermont Recipes Help, and click "AppleScript support" in the Featured Topics section. In the AppleScript support page, click the Open AppleScript Editor Help link. After a short pause, the title page of the AppleScript Editor help book appears.

2. You may have noticed that most of the search results include not only the name of the help page but also a somewhat wordier summary of the subject matter of the page. This summary is called an *abstract*. It is obviously useful to a user. When you examined the search results, you could tell at a glance what the other help pages were about, and their abstracts usually even included the name of the application in text to supplement the information conveyed by the application icon. But the AppleScript support result had no abstract, and if you didn't recognize the application icon, you would not have a clue what application this help page was about.

You should therefore always add an abstract to virtually every page in your help book. You can make Help Indexer do this for you by setting the "Generate missing summaries (slow)" option in the Help Indexer window. It chooses a sentence from the help page and uses it as the abstract. I've never tried this, because I want more control over the end result. It wouldn't work at this point, anyway, because you haven't yet written any content for the AppleScript support page.

Write your own abstract for the AppleScriptSupport.html file by adding this statement after the keywords statement:

```
<meta name="description" content="Describes the AppleScript support
in Vermont Recipes.">
```

Unfortunately, the abstract will not appear in the search results unless you register your help book programmatically. The *Apple Help Programming Guide* does not make this clear, suggesting in several places that registration is accomplished by inserting the two entries in the application's Info.plist file that you added in Step 1. Although this is correct for most purposes, it has been the case since time immemorial that you must register the help book programmatically to make the abstract show up in search results. Prior to Snow Leopard, you had to do this by calling the `AHRegisterHelpBook()` function. Now, in Snow Leopard, you can call the new `AHRegisterHelpBookWithURL()` function, or in Cocoa do it by simply instantiating a `sharedHelpManager` object using `NSHelpManager`. For the time being, do this in the `NSApplicationController.m` implementation file by adding the following statement at the end of the existing `-applicationDidFinishLaunching:` delegate method. You will revise this in Step 8 to support help abstracts under Leopard too.

```
[NSHelpManager sharedHelpManager];
```

Once again, save the AppleScriptSupport.html files, move the com.apple.helpd file to the Trash, re-index the help book, and clean the project. Instead of running the application, go to the Finder and choose Help > Mac Help. In HelpViewer's search field, pull down the search menu and choose Search All Books. Then type OSA and press Return. The search results list, as before, shows an item labeled "AppleScript support" with the Vermont Recipes application icon beside it. This time, it includes an abstract that explains exactly what this page is about.

Step 6: Add Help Buttons to Alerts, Dialogs, and Panels

Most applications include a help button in alerts, dialogs, and panels if they aren't self-explanatory. The help button is a little circle surrounding a question mark. If you design a custom panel in Interface Builder, the Interface Builder Library supplies the help button. The Print panel you implemented in Recipe 9 contains a help button by default in the lower-left corner. If you run Vermont Recipes, open the Chef's Diary, choose File > Print, and then expand the Print panel, you will see it. Click it, and you are immediately taken to Mac Help's Print dialog help page. A help button in an alert, dialog, or panel is a great convenience for the user, because it obviates the need to think up a search term or browse through the help book.

In this step, you first make the help button in the Print panel take you to a custom Vermont Recipes printing topic page instead of to the Mac Help Print dialog page. Then you implement help buttons in some of the alerts in the application.

1. First, make the standard help button in the Print panel open a custom help page for Vermont Recipes printing, instead of the Print dialog help topic it normally opens.

Start by creating a Printing topic page. As you did when you created the AppleScript support page in Step 4, simply duplicate ShoppingList.html in the Finder, rename it **Printing.html**, and change its internal title elements from Shopping lists to **Printing**. Also change the help anchor from shoppinglistpage to **printingpage**. Be sure to add a reference to this page on the title page by adding a statement like this at the end of the Featured Topics section of the Vermont Recipes Help.html file:

```
<p class="boxlink"><a href="pgs/Printing.html">Printing</a></p>
```

Now open the `DiaryDocument.m` implementation file and add this statement at the end of the main `else` branch in the `-printOperationWithSettings:error:` method:

```
[printPanel setHelpAnchor:@"printingpage"];
```

The Print panel always has a help button, so there is no separate method in `NSPrintPanel` to add a help button to the panel.

2. Next, add a help button to the alert that is displayed when the user attempts to scale the printed page to more than 100%.

Again create a new topic page. In a departure from your practice to this point, don't add a link to this topic page to the Vermont Recipes Help title page. This topic page will be accessible only from the alert's help button. Duplicate `ShoppingList.html` in the Finder and rename it **AlertPrintScaling.html**. Change its internal title elements from Shopping lists to **Cannot print larger than 100%**. The title of the alert is identical to the message text of the alert to help the user associate the help page with the alert. Also change the help anchor from `shoppinglistpage` to **alertcannotprintscaleduppage**.

Open the `DiaryPrintView.m` implementation file and add these statements to the `-alertCannotPrintScaledUp` method, just before the `return` statement:

```
[alert setShowsHelp:YES];  
[alert setHelpAnchor:@"alertcannotprintscaleduppage"];
```

There are three other alerts in the application at this point. Adding a help button to each of them is left as an exercise for the reader. The downloadable project files contain the additional help files and supporting code.

Step 7: Advanced Help Features

There are a number of more advanced features of Apple Help that are not implemented in this recipe. Complete details appear in the *Apple Help Programming Guide*. They include the following:

- QuickTime movies

Include links to QuickTime movies for show-me help.

- Internet-based content

Include links to help content that resides on a remote server, so that you can keep your help files up to date without distributing a new version of the application.

You can provide Internet-only content, Internet-primary content, or local-primary content.

- Segmented help pages

Segmented help allows you to break help content into separate topics without using multiple files.

- VoiceOver summaries

Provide spoken VoiceOver summaries that explain the relationship between cells in an HTML table.

- Exact match searching

Create word lists that enhance the relevance ranking of specified terms.

- List generation

Generate lists of links for use in indexes and “see also” sections.

- Contextual help

Implement help menu items in contextual menus, or complex help tags attached to user interface items that are displayed only when the user specifically requests them.

Step 8: Implement a Help Book for Leopard and Earlier

Until now, this recipe has been exclusively about adding a help book to an application that will run under Mac OS X 10.6 Snow Leopard. Snow Leopard introduced a new help book bundle format that does not work when an application is run under Leopard and older.

The older help book format and APIs still work under Snow Leopard, and currently most applications from third parties as well as several Apple applications use the old format so that their help books will work under both Snow Leopard and earlier versions of Mac OS X. To learn how to implement help books using the older techniques, read *Providing User Assistance With Apple Help*, a legacy document available on Apple’s legacy documentation Web site at <http://developer.apple.com/legacy/mac/library/navigation/index.html>. It covers Mac OS X 10.3 and older, but it includes material that more clearly explains some of the requirements that still apply to Mac OS X 10.4 and 10.5.

In this step, you go through the process of setting up a legacy-style help book for the Leopard version of Vermont Recipes, using the existing help files you have already created. The result will be a version of Vermont Recipes with an old-style help book that works under both Leopard and Snow Leopard.

The key to reusing the Snow Leopard help files in an old-style help book for Leopard is the fact that the project now has two targets, one for Snow Leopard and one for Leopard. In Step 1, you arranged to have the new Snow Leopard help bundle copied into the Vermont Recipes SL target, but it is not copied into the Vermont Recipes target for Leopard. To verify this, expand the Vermont Recipes and Vermont Recipes SL targets in the Xcode project window. The Copy Bundle Resources build phase in the Vermont Recipes SL target includes the `VermontRecipesHelp.help` bundle, but the Copy Bundle Resources build phase in the Vermont Recipes target does not. You will now create an old-style Vermont Recipes Help folder to be loaded into the `English.lproj` folder of the Vermont Recipes target.

Additional steps will be required. For example, the project currently contains a single `Info.plist` file for the application, named `Vermont_Recipes-Info.plist`. This won't do, because the `CFBundleHelpBookFolder` and `CFBundleHelpBookName` entries contain values that work only under Snow Leopard. You will have to create a second, separate `Info.plist` file for the Leopard target, and change the values of the help-related entries in it.

Finally, you will have to make modest revisions to the Snow Leopard help files to make them work correctly in an old-style help folder.

1. Start by creating the new Leopard help folder. You have maintained the `VermontRecipesHelp.help` bundle separately from the project. Do the same thing for the Leopard help folder. Wherever you are keeping the Snow Leopard help bundle, create alongside it a new folder and name it **Vermont Recipes Help**. You will arrange shortly to have it copied into the `English.lproj` folder of the built Leopard target.

Then drag copies of the entire contents of the `English.lproj` folder in the Snow Leopard help bundle into the top level of the new Vermont Recipes Help folder.

Next, because old-style help books don't support shared graphics files, drag copies of the four application icon files from the `Shared` folder in the Snow Leopard help bundle into the `gfx` subfolder in the new Vermont Recipes Help folder.

Old-style help books don't have their own `Info.plist` files, so leave the Snow Leopard help bundle's `Info.plist` file alone.

Now build a new index file for the Leopard version of Vermont Recipes Help. Open the new Vermont Recipes Help folder, and drag the existing `Vermont Recipes.helpindex` file to the Trash. Drag the Vermont Recipes Help folder onto

the Help Indexer utility. You now find in the Vermont Recipes Help folder a new file named Vermont Recipes Help.helpindex.

2. Add the new Vermont Recipes Help folder to the project. This time, instead of designating it to be copied into the Vermont Recipes SL target, designate it not to be copied at all. It belongs in the English.lproj folder of the built target's Resources folder, and you will shortly arrange to copy it there by creating a new Copy Files build phase dedicated to this one folder.

Select the Resources group in the Groups & Files pane of the Xcode project window and choose Project > Add to Project. In the Save panel, select the new Vermont Recipes Help folder you just created and click Add. In the next sheet, leave the "Copy items into destination group's folder (if needed)" checkbox deselected. Select the "Create Folder References for any added folders" radio button. And in the Add To Targets table, deselect both the Vermont Recipes target and the Vermont Recipes SL target. Then click Add.

You now see two help books in the Resources group: Vermont Recipes Help and VermontRecipesHelp.help. To verify that the new Vermont Recipes Help folder will not be copied into either target, expand the Vermont Recipes and Vermont Recipes SL targets in the Xcode project window again. The Copy Bundle Resources build phase in the Vermont Recipes SL target still includes the VermontRecipesHelp.help bundle, and it does not include the new Vermont Recipes Help folder for Leopard. The Copy Bundle Resources build phase in the Vermont Recipes target does not include either help book.

3. You do, of course, need to get the new Vermont Recipes Help folder into the built Vermont Recipes target for Leopard, but it needs to be placed in the English.lproj folder in the application package's Resources folder. To accomplish this, create a new Copy Files build phase in the Vermont Recipes target.

Expand the Vermont Recipes target, and select the existing Copy Files build phase. Then choose Project > New Build Phase > New Copy Files Build Phase. In the General pane of the Copy Files Phase dialog that opens, leave the Destination pop-up menu set to Resources, and enter **English.lproj** in the Path text field. Close the Copy Files Phase dialog. Then drag the Vermont Recipes Help folder from the Resources group and drop it beneath the new Copy Files build phase in the Vermont Recipes target.

To verify that this is working as intended, go to the Xcode project window, and, using the Overview pop-up menu, set the active target to Vermont Recipes. Choose Build > Clean, and then choose Build > Build. Finally, find the built Vermont Recipes application in the Debug subfolder of the build folder in the Vermont Recipes project folder, and choose Open Package Contents in the contextual menu. Open the Contents folder and then the Resources folder, and

note that the `VermontRecipesHelp.help` bundle for Snow Leopard is not there. Open the `English.lproj` folder, and verify that the Vermont Recipes Help folder for Leopard is there, right where it belongs, along with all of its help files.

There is no point in running the Leopard version of the application yet, however, because you haven't yet arranged to place the Leopard version of the application's `Info.plist` file in the built Leopard target.

4. The procedure for creating a modified `Info.plist` file for the Leopard target is similar. You will end up with two separate `Info.plist` files in the project, with slightly different names, one of which will be built into the Leopard target as its `Info.plist` file and one of which will be built into the Snow Leopard target as its `Info.plist` file. Each `Info.plist` file will contain appropriate `CFBundleHelpBookFolder` and `CFBundleHelpBookName` values.

Start by renaming the existing `Vermont_Recipes-Info.plist` file as **Vermont_Recipes_SL-Info.plist**. Do this by selecting it in the Xcode project window and, using the contextual menu, choosing `Rename`. After it is renamed in Xcode, the name of the file in the Finder and in the Vermont Recipes target's build settings is also automatically renamed. To verify the latter, open an Info window on the Vermont Recipes SL target, select the Build tab, and find the `Info.plist File` (or `INFOPLIST_FILE`) key in the Packaging section. You see that its value is now `Vermont_Recipes_SL-Info.plist`.

Now make a duplicate copy of the file. Find the renamed `Vermont_Recipes_SL-Info.plist` file in the project folder in the Finder, duplicate it, and name the copy **Vermont_Recipes-Info.plist**. Drag it from the Finder into the Xcode project window and drop it just below the `Vermont_Recipes_SL-Info.plist` file. In the sheet that opens, select the "Recursively create groups for any added folders" radio button, deselect both the Vermont Recipes target and the Vermont Recipes SL target checkboxes, and then click `Add`.

Select the Vermont Recipes target, open its Info window, select the Build tab, and change the value of the `Info.plist File` (or `INFOPLIST_FILE`) key in the Packaging section to **Vermont_Recipes-Info.plist**. Be sure to make this change in both the Debug and Release configurations in the Vermont Recipes target's Info window.

Finally, double-click the new `Vermont_Recipes-Info.plist` file in the Groups & Files pane to open it. Change the value of the `CFBundleHelpBookFolder` entry to **Vermont Recipes Help**, and change the value of the `CFBundleHelpBookName` entry to **Vermont Recipes Help** as well.

5. You're now ready to build and run the Vermont Recipes target for Leopard.

First, perform the ritual of moving the `com.apple.helpd` file to the Trash and re-indexing the help book. The Overview pop-up menu in the Xcode project window is still set to Vermont Recipes as the active target. Choose Build > Clean and then build and run the application. Once it's running, choose Help > Vermont Recipes Help. By golly, the Vermont Recipes Help title page appears in Help Viewer.

6. Play around with the help book and catalog any problems you encounter. It was too much to hope that this would work perfectly, and you will have to do a modest amount of cleanup work.

First, none of the application icons appear, but only placeholders. The reason is simple: The icon path in each help file refers to the Shared folder alongside the `English.lproj` folder in the Resources folder of the `VermontRecipesHelp.help` bundle, but the Leopard help book doesn't have a Shared folder—you moved all the icon files into the `gfx` subfolder of the Vermont Recipes Help folder. To cure this problem, revise the icon paths in each help file in the Vermont Recipes Help folder by replacing Shared with **gfx** and specifying the new nesting level. For example, in the `AboutVermontRecipes.html` file and all the other topic files, change `../Shared/` to `../gfx`. In the Vermont Recipes Help.html title page, change `../Shared` to **gfx**.

Second, the “Reveal Vermont Recipes in the Finder” AppleScript link in the About Vermont Recipes topic page does not work. It relies on the `x-help-script:` URL, which depends on a bundle identifier that the Leopard version of the help book does not contain due to the absence of an `Info.plist` file. The old pre-Snow Leopard `help:runscript` protocol still works in Snow Leopard, so switch to that. Change the AppleScript element in the `AboutVermontRecipes.html` file to this:

```
<p><a href="help:runscript='Vermont%20Recipes%20Help/scrip/  
  OpnAppBndID.scpt'%20string='reveal,com.quecheesoftware.Vermont-  
  Recipes'">Reveal Vermont Recipes in the Finder  
  <span class="linkarrow"></span></a></p>
```

Third, the Related Topics links in the About Vermont Recipes page don't work. These suffer from the same problem as the AppleScript link: They depend on a help bundle identifier. Fix this problem by using a hard-coded link. Replace the two Related Topics links at the bottom of the `AboutVermontRecipes.html` file with this:

```
<p class="linkinternal"><a href="Recipes.html">Recipes  
  <span class="linkarrow"></span></a></p>  
<p class="linkinternal"><a href="ChefsDiary.html">Chef's Diary  
  <span class="linkarrow"></span></a></p>
```

Finally, the Vermont Recipes Help menu item in Help Viewer's Library menu doesn't show the application icon, and it ends in *.html*. Both problems are easily fixed by editing the `AppleTitle` and `AppleIcon` meta tags in the Vermont Recipes Help.html title page. Change them to this:

```
<meta name="AppleTitle" content="Vermont Recipes Help"/>
<meta name="AppleIcon" content="../../gfx/VRApplicationIcon016.png"/>
```

7. There is one more bit of code you might want to add now, just in case you ever decide to open specific help book pages programmatically. Most of the Apple Help features you have added to the application so far work correctly when you register the help book in the Info.plist file. However, if you want to open help pages programmatically, you must register the help book programmatically, too. You did this for Snow Leopard in Step 5 when you added `[NSHelpManager sharedHelpManager]` to the `-applicationDidFinishLaunching:` method in `NSApplicationController.m`. For Leopard, you must call the `AHRegisterHelpBook()` function instead. Revise the end of the `-applicationDidFinishLaunching:` method so it looks like this:

```
if (floor(NSAppKitVersionNumber) > NSAppKitVersionNumber10_5) {
    [NSHelpManager sharedHelpManager];
} else {
    CFBundleRef applicationBundleRef = NULL;
    CFURLRef applicationBundleURL = NULL;
    FSRef applicationBundleFSRef;
    applicationBundleRef = CFBundleGetMainBundle();
    if (applicationBundleRef) {
        applicationBundleURL =
            CFBundleCopyBundleURL(applicationBundleRef);
        if (applicationBundleURL) {
            if (CFURLGetFSRef(applicationBundleURL,
                &applicationBundleFSRef)) {
                AHRegisterHelpBook(&applicationBundleFSRef);
            }
        }
    }
    if (applicationBundleURL) CFRelease(applicationBundleURL);
}
```

The Leopard branch is essentially identical to the code set out in the “Help Book Registration” section of the *Apple Help Programming Guide*, except that this method uses `AHRegisterHelpBook()` instead of the Snow Leopard-only `AHRegisterHelpBookWithURL()` function shown in the *Guide*.

To compile it successfully, you must import the Carbon framework near the top of the `VRApplicationController.m` implementation file like this:

```
#import <Carbon/Carbon.h>
```

You must also add the Carbon framework to the Linked Frameworks subgroup of the Frameworks group in the project window. Select the Linked Frameworks subgroup. Using the contextual menu, choose **Add > Existing Frameworks**, select `Carbon.framework` in the sheet, and click **Add**. You must also open the Vermont Recipes and Vermont Recipes SL targets and make sure `Carbon.framework` is included under **Link Binary With Libraries** for the Vermont Recipes and Vermont Recipes SL targets.

That's it. You now have a help bundle for Snow Leopard and a help folder for Leopard, and both help books look and work the same.

Step 9: Build and Run the Application

You've already built and run the application, both for Snow Leopard and for Leopard. Apple Help is working just as it should in both environments.

Step 10: Save and Archive the Project

Quit the running application. Close the Xcode project window, discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like *Vermont Recipes 2.0.0 - Recipe 11.zip*. The working Vermont Recipes project folder remains in place, ready for Recipe 12.

Conclusion

You now have three of the four single-topic recipes under your belt: You've added printing support, a preferences window, and Apple Help to Vermont Recipes. In the next recipe, you will take on the last single topic, adding AppleScript support.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 11.

Class Reference and Protocol Documents

NSHelpManager Class Reference

NSUserInterfaceItemSearching Protocol Reference

NSBundle Class Reference

NSBundle Additions Reference

General Documentation

Online Help

Apple Help Programming Guide

Technical Q&A QA1022: Where should I install my help book, and how does Help Viewer locate it?

hiutil(1) man page

Bundle Programming Guide

Providing User Assistance With Apple Help (legacy)

SimpleHelp (legacy sample code)

Add AppleScript Support

This is the last of a series of recipes dealing with single topics. In this recipe, you add AppleScript support to the Vermont Recipes application using Apple’s Cocoa Scripting API. Cocoa Scripting is a big topic. You will exercise most of its major features here, but there isn’t room to cover everything. At the end of this recipe, you will find some pointers to guide you through the remaining areas.

AppleScript is a scripting language that focuses on interapplication communication. Calling it a scripting language emphasizes that it is accessible by people who don’t consider themselves to be programmers. It invites them to write scripts using what look very much like plain English sentences. Although scripters don’t have to be programmers, they should have an interest in automating some of the tasks that they perform on their computers, either to standardize complex and error-prone operations or to avoid having to perform boring, repetitive tasks repeatedly by hand. AppleScript works with Apple events at the system level to let scripters control many aspects of the operating system and system-level utilities. It also gives scripters direct control over many applications, whether from Apple or from third parties. Any application that supports AppleScript can be integrated with other scriptable applications into a powerful workflow engine to accomplish tasks that no single application is able to perform by itself. Adding AppleScript support to your application will increase its appeal and widen its market.

This recipe assumes that you have some familiarity with AppleScript. Indeed, familiarity with AppleScript is essential in order to get your application’s AppleScript

HIGHLIGHTS

- Turning on AppleScript support*
- Creating a terminology dictionary (scripting definition file)*
- Including the Standard Suite*
- Adding a custom suite*
- Adding the Text Suite*
- Adding HTML documentation to the dictionary*
- Extending and adding classes*
- Adding properties, elements, types, and commands*
- Getting and setting properties*
- Supporting undo and redo*
- Supporting Standard Suite commands*
- Adding verb-first custom commands*
- Adding object-first custom commands*

support right. AppleScript is a mature and well-established technology, going back to the early 1990s. There is a large and established community of AppleScript users, and they have traditions and strongly held expectations regarding the design of an application's terminology dictionary. You should offer AppleScript support with your application, and when you do, you must meet the community's expectations by providing a proper AppleScript terminology dictionary.

There are many good AppleScript books in print to help get you started. At this writing, the most recent is the book that I coauthored with Apple's AppleScript product manager, Sal Soghoian, *Apple Training Series: AppleScript 1-2-3* (Peachpit Press, 2009).

For programmers, supporting AppleScript in an application was once a daunting and highly technical task. Although AppleScript was included in the initial migration of the old Classic system into Mac OS X, it took longer than most Apple technologies to become fully integrated into the Cocoa frameworks. AppleScript's transition to Cocoa was largely complete by the time of Mac OS X 10.5 Leopard, and supporting AppleScript is now reasonably easy. There is ample Apple documentation for Cocoa Scripting. However, despite tremendous improvements over the last few years, it can still be difficult to follow. I hope that this recipe puts Cocoa Scripting together in a way that will speed your application's adoption of comprehensive AppleScript support.

There is more ground to cover than room to cover it, so some of the AppleScript features implemented in this recipe are posed as challenges to the reader. All of them are fully implemented in the downloadable project file for Recipe 12.

Step 1: Create a Terminology Dictionary and Add the Standard Suite

AppleScript is unique in the world of programming and scripting in that its terminology can be redefined by every application that supports it. The core AppleScript language consists of very few classes and commands. Almost every scriptable application adds additional classes and commands and extends or alters the meaning of core AppleScript terms. To maintain consistency and usability, developers of scriptable applications are expected to exercise self-discipline by following established guidelines regarding terminology and syntax.

Because the language is extensible by every application developer, applications inevitably depart from these guidelines to a greater or lesser degree. Fortunately, the same file in which a developer defines an application's AppleScript terminology also serves as the application's AppleScript reference manual. It is known as the application's

terminology dictionary, and you can look up the application's terminology in it just as you would look up a word in a printed dictionary.

The dictionary lists all of the terminology suites and classes that a scriptable application supports, including each class's unique elements and properties. It also lists all of the commands the application supports, including their parameters and return values, as well as special data types defined by the application. In addition, the dictionary contains brief descriptions of each class, element, property, command, parameter, return value, and type. In recent releases of Mac OS X, the dictionary can even include HTML documentation and example scripts to enhance a scripter's ability to script the application. Since a developer must write the terminology dictionary in order to implement AppleScript support, the documentation gets written at the same time almost automatically as part of the development process. A scripter can open the dictionary and view it using any script editor, such as Apple's AppleScript Editor or Late Night Software's Script Debugger, while using that same editor to compose the script (**Figure 12.1**).

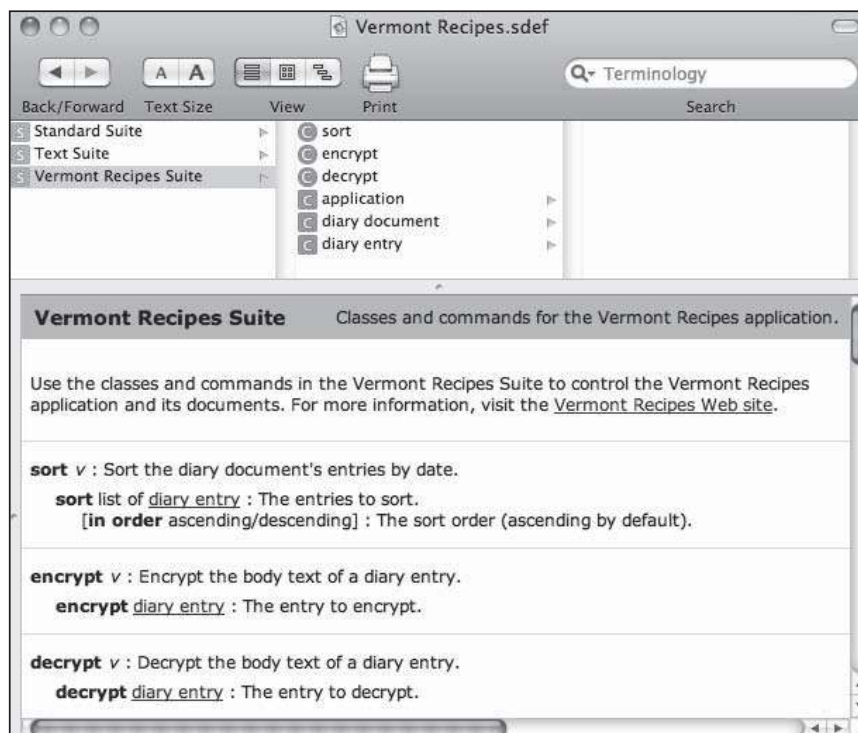


FIGURE 12.1
The Vermont Recipes dictionary viewed in AppleScript Editor.

In the early years of AppleScript, even into the Mac OS X era, dictionaries took the form of an *aete resource*. Later, they took the form of twin files known by their file extensions as *scriptSuite* and *scriptTerminology* files. You still see many of these in current applications, such as TextEdit, when you look inside the application package at its Resources folder. However, since Mac OS X 10.4 Tiger, Cocoa Scripting has been able to read the current preferred form of dictionary file, the *scripting definition*, or *sdef*, file. These are XML files, so it is easy to write them in any XML editor or, for that matter, in any plain old text editor. You can find examples of sdef files in

several current Apple applications, including Address Book, Aperture, AppleScript Utility, iCal, iChat, iWeb, and QuickTime Player. You can also find them in many third-party applications, including Interarchy, OmniFocus, OmniPlan, Path Finder, QuicKeys, Smile, and Yojimbo. If you are developing for Leopard or newer, you should use sdef files.

In this step, you create the Vermont Recipes sdef file with some initial content, and then you turn on AppleScript support and point the application to the sdef file by adding two entries to the Info.plist file. You may be surprised to find that the application is scriptable even with such little effort.

1. Leave the archived Recipe 11 project folder in the zip file where it is, and open the working Vermont Recipes subfolder. Increment the version in the Properties pane of the Vermont Recipes target's information window from 11 to 12 so that the application's version is displayed in the About window as 2.0.0 (12).
2. Now create the sdef file. It is a plain text file, and you can create it in Xcode as part of the Vermont Recipes project. In the Xcode project window, select the Resources group. Choose File > New File, select Other in the left pane, select Empty File in the upper-right pane, and click Next. Name it **Vermont Recipes.sdef**, select both the Vermont Recipes and Vermont Recipes SL targets, and click Finish. Place it in the Resources group.
3. Open the new Vermont Recipes.sdef file. By default, it opens in a terminology dictionary viewer that looks just like the dictionary viewer in AppleScript Editor, but you can't edit it. Use the contextual menu to choose Open As > Plain Text File. This opens the new file for editing as a plain text file in the editing pane of the main Xcode project window. If you prefer to edit it in a separate window, double-click it.
4. Enter the header information required at the top of every sdef file and a bare-bones dictionary element, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dictionary SYSTEM
    "file://localhost/System/Library/DTDs/sdef.dtd">
<dictionary xmlns:xi="http://www.w3.org/2003/XInclude">
    <xi:include href="file:///System/Library/ScriptingDefinitions/
        CocoaStandard.sdef" xpointer="xpointer(/dictionary/suite)"/>
</dictionary>
```

Disregarding the soft line breaks imposed by the dimensions of the printed page, these are five lines, consisting of the XML declaration, the identification of the DTD file defining the format, the XML tag defining the beginning of the

dictionary element, an **XInclude** element within the **dictionary** element, and the tag closing the **dictionary** element.

Since Leopard, Cocoa Scripting allows you to use **XInclude** (XML Inclusions) elements to reference external XML elements. In this case, it includes the official **CocoaStandard.sdef** file located in the System Library's **ScriptingDefinitions** folder in Leopard and Snow Leopard. The **CocoaStandard.sdef** file defines the **AppleScript Standard Suite**, a suite of commands and classes supported by virtually all scriptable applications. It includes the **application**, **document**, and **window** classes and several commands, such as **open**, **delete**, and **make**, as well as some common types. For now, the Standard Suite is the only suite that can be included in an application's dictionary using the **XInclude** mechanism.

Later in this recipe, you will delete the **XInclude** element and instead copy and paste the Standard Suite in its entirety into the **Vermont Recipes.sdef** file, so that you can make a modification to it. It is common practice to do this in order to delete terms your application does not support or to make other changes. Alternatively, the documentation indicates that you can delete subelements of included elements using the XML **XPointer** standard described at <http://www.w3.org—xinclude>. For now, however, the **XInclude** element in the **dictionary** element will serve your purposes.

5. Now turn on AppleScript support and point the application at your new **sdef** file. This requires making two changes to the **Info.plist** files. Since you now have two **Info.plist** files, you must make these changes in both of them.

To turn on AppleScript support, add an entry for the **NSAppleScriptEnabled** (Scriptable) key if it isn't already present, and in Xcode's property list editor window, select the checkbox to set the value to true. To identify your new **sdef** file, add an entry for the **OSAScriptingDefinition** (Scripting definition filename) key and set the value to **Vermont Recipes.sdef**.

6. It is worth getting a little immediate gratification at this point. Build and run the application, create or open a diary document, and save it under the name *Chef's Diary*. Then launch AppleScript Editor. In Snow Leopard, it is located in the **/Applications/Utilities** folder. In Leopard and earlier it was called **Script Editor** and was located in the **/Applications/AppleScript** folder.

Then enter this script in a new AppleScript Editor editing window:

```
tell application "Vermont Recipes"
    get first document whose name begins with "Chef"
end tell
```

You don't have to click the **Compile** button, although you can if you just want to check the script's syntax. Click **Run** to compile and run the script in a single step.

In the Result pane at the bottom of the window, you see the result, a reference to document "Chef's Diary" of application "Vermont Recipes" (Figure 12.2).

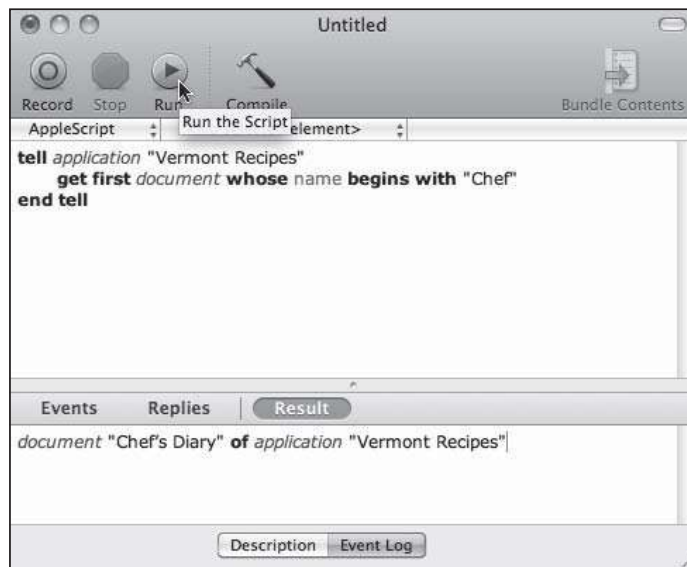


FIGURE 12.2 AppleScript Editor after running a simple script.

This script and others work now because several commands and classes are included in the Standard Suite, and the underlying code to support them is included in the Cocoa frameworks. As you might have guessed, the code supporting the **document** class is `NSDocument`. You probably haven't previously encountered the classes that support various commands, but they're in `Foundation`, all declared as subclasses of `NSScriptCommand`.

Congratulations! You've written your first scriptable application. The capabilities you have just experimented with are part of an application's built-in support for AppleScript and part of the Standard Suite of AppleScript terminology that you made available to the application by setting up the `sdef` file. Full AppleScript support is not this easy, of course. Many things that a scripter expects to be able to do are not yet working. You'll add much more in the coming steps.

7. Before continuing, read the Standard Suite in its entirety if you aren't already an experienced scripter. It is important to understand the terminology it provides, because you won't have to duplicate any of it in the custom terminology suite that you will start writing in Step 2. In addition, you will begin to absorb the characteristic style of AppleScript terminology, which has been well established over nearly two decades of use. Even the descriptions and instructional text in the Standard Suite establish a writing and punctuation style that you should follow in your own suite for the sake of consistency. In the AppleScript community, inconsistent terminology, style, and punctuation will raise questions about the quality of your AppleScript implementation.

A good way to read the application's dictionary during development, including the Standard Suite, is to find the `Vermont Recipes.sdef` file in the Resources group in the Vermont Recipes Xcode project window. Select it, and then use

the contextual menu on it to choose Open As > AppleScript Dictionary. Then double-click it if you want to open it in a separate dictionary viewer window. It is even easier to drag the file to the AppleScript Editor application icon and drop it. The dictionary opens immediately in AppleScript Editor, and you won't have to use Xcode's contextual menu to view it as an editable text file again when you go back to editing it. You will find it useful to use either of these techniques throughout the remainder of this recipe to verify that your additions to the Vermont Recipes.sdef file look the way you want them to look.

Step 2: Add the Vermont Recipes Suite and Extend the Application Class With a New Property

The terminology suites supplied by Apple—the Standard Suite, and the Text Suite that you will add to the Vermont Recipes dictionary in Step 4—provide a range of terminology that is useful in almost every application. But most applications benefit from additional, custom terminology that scripters can use to automate the unique capabilities of a specific application. You add custom terminology to an application's dictionary by adding one or more custom application suites to the sdef file. They are typically named after the application. In this recipe, you will add the Vermont Recipes Suite, and in it you will extend the Standard Suite's `application` class and add a new `terminology version` property to it.

Apple documents the sdef format in complete technical detail in the `sdef(5)` man page. You will find it in the Xcode documentation window by entering `sdef(5)` in the Search field or by searching the Mac OS X 10.6 Core Library for the *Mac OS X Man Pages* and selecting `sdef(5)` in Section 5. Alternatively, choose Help > Open man Page in Xcode and open the `sdef(5)` man page by name; however, this version is not as well formatted. If you're a glutton for punishment, open it in Terminal by entering `man 5 sdef` and pressing Return. It is important that you master this document. It will help you to write proper sdef files, and it will be invaluable in tracking down errors in your sdef files. You should also read the "Scripting" section of the *Mac OS X Leopard Developer Release Notes: Cocoa Foundation Framework*. Unfortunately, at this writing they are missing in action due to Apple's overzealous weeding out of "legacy" documentation. A number of improvements were added to the sdef format in Mac OS X 10.5 Leopard, and they are documented only in the release notes.

Be prepared for occasional disappointment as you edit the sdef file. If you make a typographical or syntax error that renders the file unreadable by the sdef parsing

code, you will see this cryptic message in an otherwise empty dictionary viewer: “Nothing to see here; move along.” If you try to run a script against an application that contains a bad sdef file, AppleScript Editor presents an alert telling you that the application has a corrupted dictionary. You may find some more informative information about the error in the Debugger Console. Read the *Cocoa Scripting Guide* to learn about debugging your sdef file.

In addition to understanding the technical rules regarding the sdef file format, it is important to learn how scripters expect your application’s terminology to be designed. This book is not about how to write scripts or design terminology dictionaries, but you should be aware that you won’t get any respect from the AppleScript community if your design is clumsy and your terminology is awkward. The best practices are described at length in Apple’s *Technical Note TN2106: Scripting Interface Guidelines*, popularly known as the *SIG*. This is a document that you should master before you design your application’s terminology dictionary. Also, there is still value in a much older article that I coauthored with one of AppleScript’s great design gurus, Cal Simone, *The AppleScript Scorecard Guidelines*, MacTech Magazine, vol. 14, no. 2 (1998), available online at <http://www.mactech.com/articles/mactech/Vol.14/14.02/AppleScriptScorecard/index.html>.

In this step, you create the Vermont Recipes Suite. Over the course of this recipe, you will add a variety of custom classes, elements, properties, commands, parameters, and types to it. In the process, you will learn how to use most of the techniques made available through the Cocoa Scripting API.

Whenever you make changes to an application’s sdef file, quit AppleScript Editor and relaunch it before viewing your dictionary or running scripts against it. AppleScript Editor caches dictionary information, and if you don’t quit it first, you will see stale information when you use it. Even the experts among us forget to do this, and it can lead to lots of wasted time chasing imaginary problems.

1. Add the suite definition for the Vermont Recipes Suite to the sdef file. Enter this element at the end of the **dictionary** element, before the closing `</dictionary>` tag:

```
<suite name="Vermont Recipes Suite" code="VRdy"
      description="Classes and commands for the
      Vermont Recipes application.">
</suite>
```

The wording and style of the **description** attribute you provided for the Vermont Recipes Suite echo Apple’s **description** attribute in the Standard Suite. It is in sentence case with a period at the end. Almost all of your **description** attributes should be written in this style in order to maintain consistency with the Standard Suite’s style.

The `code` attribute is what is known as a *four-character code*. Apple reserves to itself all codes consisting wholly of lowercase letters and spaces, so you should include at least one uppercase letter in all alphabetic codes that you make up for your custom elements. You should use codes consisting of lowercase characters and spaces only when you use established Apple codes to reuse standard elements defined by Apple. Your codes should be unique within your application, but it doesn't matter if they are the same as codes used in other third-party applications.

Be sure to check every code you invent against the lists of existing Apple codes in the *AppleScript Terminology and Apple Event Codes Reference*. Avoid duplicating any of those existing codes. They are case sensitive, so you can use the same characters with different capitalization. Note that Apple encourages you to reuse Apple's codes when you add features to your dictionary that are intended to use existing Apple terminology, such as the `pnam` code for a `name` attribute.

2. Before adding additional classes and commands to the Vermont Recipes Suite, you should take care of a minor problem now. If you were to build and run the application and then run a script addressed to the Vermont Recipes application, you would see a warning like this in the Debugger Console: "2010-01-08 09:55:32.866 Vermont Recipes[20264:a0f] .sdef warning for argument 'FileType' of command 'save' in suite 'Standard Suite': 'saveable file format' is not a valid type name." It would be a good idea to suppress this warning.

The warning is presented because the Standard Suite you included in the `sdef` file defines the `save` command with an `as` property of type `saveable file format`, but the Standard Suite does not define a type of that name. The *Mac OS X Leopard Developer Release Notes: Cocoa Foundation Framework* explain that application developers are expected to define a type of that name themselves to specify the types of files the application can save. This recipe does not cover the `save` command, so until you define it yourself, you should define a dummy type to suppress the warning.

Insert this `enumeration` element at the top of the new Vermont Recipes Suite:

```
<!-- suppress console warning for missing "saveable file format" type -->
<enumeration name="saveable file format" code="savf" hidden="yes">
  <enumerator name="dummy" code="VTdm"
    description="A dummy file format."/>
</enumeration>
```

Just above this element, you see how to insert comments in `sdef` files. In the element, you use the `hidden` attribute to prevent the element from appearing in the human-readable dictionary. The `hidden` attribute is particularly useful for allowing obsolete terms in older scripts to continue to work with a newer version of your application, while removing the old terms from the human-readable

dictionary in favor of newer terminology. An **enumeration** element contains any number of **enumerator** subelements, but you need only one here because this enumeration is not meant to be used by scripters.

3. Next, extend the Standard Suite's **application** class so that you can begin adding custom properties and commands to the Vermont Recipes application.

Until Leopard, you would have done this by redefining the **application** class in your application's suite, but using the same code, **capp**:

```
<class name="application" code="capp"
    description="The Vermont Recipes top-level scripting object."
    inherits="application">
    <cocoa class="NSApplication"/>
</class>
```

You would have reused Apple's all-lowercase Standard Suite code for the **application** class, **capp**, because the Vermont Recipes Suite's **application** class inherits from that class. Using the **inherits** attribute with the name of the **application** class makes the inheritance relationship clear. The **cocoa** subelement tells Cocoa that the **application** class is supported in code by Cocoa's **NSApplication** class.

Starting with Leopard, however, you should instead use the new **class-extension** element. Insert this element in the Vermont Recipes Suite just after the dummy **saveable file format** enumeration:

```
<class-extension extends="application"
    description="The Vermont Recipes top-level scripting object.">
</class-extension>
```

As you see, you omit the **name**, **code**, and **inherits** attributes and the **cocoa** subelement, relying instead on the new **extends** attribute to incorporate all of them from the Standard Suite's **application** class.

4. Now you can begin adding custom properties to the **application** class. First, define a new **terminology version** property in the **sdef** file. Add this **property** element in the new **application class-extension** element:

```
<property name="terminology version" code="VRtv" type="integer" access="r"
    description="The version of the application's terminology
    dictionary.">
    <cocoa key="terminologyVersion"/>
</property>
```

Properties have **type** and **access** attributes. This property is typed as an integer, so the dictionary's terminology version might be 1, 2, or 3, and so on, as the application is revised over time. You could have made it a text property to accommodate

a more complex versioning scheme, but then it would be harder to compare older and newer versions. By default, a property's `access` attribute is `rw` for read-write, but scripters should not be allowed to change the terminology version number. Here, you therefore make it `r` for read-only. The `cocoa` subelement for properties takes a `key` attribute, which is given as `terminologyVersion` here. You will implement this in code as a `-terminologyVersion` accessor method shortly. The value of the `key` attribute must be exactly the same as the name of the method, because Cocoa Scripting uses key-value coding (KVC) to determine which method to call.

5. Finally, you have to write some supporting code to implement the `terminologyVersion` property.

Developers employ a variety of styles for adding AppleScript support code to an application.

At one extreme, they might choose to intermix all of the AppleScript support methods with the other methods in the application's existing classes.

At the other extreme—which I favor—developers create separate code files for as much of their AppleScript support as possible. Many of the separate files contain categories on existing classes in the application. Doing it this way allows you to place most of your application's AppleScript support in a separate AppleScript Support group in the project window's Groups & Files pane. It is not uncommon for an application's AppleScript support to be written by a separate team, and this helps to keep the teams out of each other's way. It is also a useful technique when AppleScript support is added to an existing nonscriptable application. Inevitably, however, some code must be added to existing classes. For example, instance variables can't be added to a class in a category, so if you add accessors in a category, you must add any supporting instance variables in the existing class.

You'll use the latter approach in Vermont Recipes. Start by creating a new AppleScript Support subgroup in the Classes group in the Xcode project window's Groups & Files pane. Select the existing Views & Responders subgroup, if that's at the bottom of the Classes group, and use the contextual menu to choose `Add > New Group` below it; name the new group **AppleScript Support**.

Then select the new AppleScript Support group, and, again using the contextual menu, choose `Add > New File` and go through the familiar process of creating a new pair of Cocoa source files. In the New File dialog, select Cocoa Class in the source list on the left, select "Objective-C class" in the upper-right pane, choose subclass of NSObject in the pop-up menu below that, click Next, name the file **VRApplicationController+VRAppleScriptAdditions.m**, select the checkbox to create a header file with the same name, select both the Vermont Recipes and Vermont Recipes SL targets, click Finish, and add your customary information to the top of both files. The name follows the pattern you have adopted for

naming category files, starting with the name of the base class and, following a plus (+) sign, adding the category name.

6. Write the category code to support the new terminology version property. For AppleScript properties, or to-one relationships, you write simple accessor methods. If it is a read-only property, you need only a getter. If it is a read-write property, you need a getter and a setter. The terminology version property is read-only, so write a getter, `-terminologyVersion`.

Take a moment to reflect on the name of this getter. Remember that a getter and, for read-write variables, a setter should be key-value coding (KVC) compliant. You can use any of a few naming variants, but the most common are described in the documentation as `-<key>` and `-set<Key>:`. In this case, the key is `terminologyVersion`, so the getter is `-terminologyVersion`. If it were a read-write variable, the setter would be `-setTerminologyVersion:`. Cocoa Scripting is based on KVC, so KVC compliance is essential, and in addition it usually gives you automatic support for Cocoa bindings, which are discussed in Recipe 14. Remember this usage of `<key>` and `<Key>`, because it will become even more important in Steps 5, 8 and 9, where you implement AppleScript elements, or to-many relationships, using a closely related technique.

In the new category header file, import the base class's header file by adding an import directive to the list of imported files, like this:

```
#import "VRApplicationController.h"
```

Change the `@interface` directive to this, without any curly braces:

```
@interface VRApplicationController (VRAppleScriptAdditions)
```

Also change the `@implementation` directive in the category implementation file to this:

```
@implementation VRApplicationController (VRAppleScriptAdditions)
```

Back in the header file, declare the `-terminologyVersion` accessor method:

```
#pragma mark ACCESSOR METHODS
```

```
- (NSNumber *)terminologyVersion;
```

Define it in the category implementation file like this:

```
#pragma mark ACCESSOR METHODS
```

```
- (NSNumber *)terminologyVersion {  
    return [NSNumber numberWithInt:APPLESCRIPT_TERMINOLOGY_VERSION];  
}
```


Typically, you would implement an accessor method by declaring an instance variable named `terminologyVersion`, initializing it, and returning its value. Because this is a category on `VRApplicationController`, you would have to declare it in the header file for that class. However, this property is not in fact variable within any one version of the application, so it is cleaner simply to define it. At the top of the category implementation file, just before the `@implementation` directive, insert this definition:

```
#define APPLESCRIPT_TERMINOLOGY_VERSION    1
```

When you upgrade the Vermont Recipes application at some future date, you will want to remember to increment the version number to 2 if you modify the AppleScript terminology.

7. Whenever you implement AppleScript terminology that is part of the `application` class, as the `terminology version` property is here, you have to include some code to help Cocoa Scripting find your implementation. This is because developers don't ordinarily subclass `NSApplication`. More typically, they create a separate class to act as `NSApplication`'s delegate, as you did here with `VRApplicationController` in Recipe 5. Since you did not subclass `NSApplication` or add a category to it, you had to write the `-terminologyVersion` method in the delegate, `VRApplicationController`. Nevertheless, the `sdef` file specifies `NSApplication` as the Cocoa class for the AppleScript application class, through the `class-extension` element you just added.

Cocoa Scripting accommodates this common design pattern by providing the `-application:delegateHandlesKey:` delegate method. All you have to do is implement it at the end of the `VRApplicationController.m` implementation file, after the existing `-applicationDidFinishLaunching:` delegate method, like this:

```
- (BOOL)application:(NSApplication *)sender
    delegateHandlesKey:(NSString *)key {
    if ([key isEqualToString:@"terminologyVersion"]) {
        return YES;
    }
    return NO;
}
```

This tells Cocoa Scripting that any script referring to the `terminology version` property is handled by an accessor method named `terminologyVersion` in the application delegate. You long ago connected `VRApplicationController` as the application delegate in Interface Builder. Later, when you add additional application properties to the `sdef` file, you will have to remember to add their keys to this delegate method. Forgetting this is a common mistake, but your scripts accessing the property won't work until you do it.

8. At this point, you have a working `terminology version` property. To test it, build and run the application. Don't forget to quit AppleScript Editor and relaunch it before running any scripts, to clear its existing terminology cache so that it can recognize the new property. Then run this test script:

```
tell application "Vermont Recipes"
    get terminology version
end tell
```

In the Result pane, it returns 1. Try running another script:

```
tell application "Vermont Recipes"
    get properties
end tell
```

It returns this AppleScript properties record: `{version:"2.0.0", name:"Vermont Recipes", frontmost:false, class:application, terminology version:1}`. The `version`, `name`, and `frontmost` properties are defined in the Standard Suite's application class and inherited by Vermont Recipes. AppleScript supplies the `class` property. The `terminology version` property, with a value of 1, is your own creation.

9. This could be the end of Step 2, but there are some refinements you should add before you move on.

First, you will find it a blessing later if you take steps now to add some debugging support. Cocoa Scripting involves a number of unfamiliar constructs, and debugging can be difficult. You should follow the instructions in Apple's SimpleScripting 1.1 example code series to include an SLOG macro in your scripting methods.

Create a new source file in the Vermont Recipes project, naming it **AppleScriptLog.h**, and place it at the top of the Classes group in the project window's Groups & Files pane. It doesn't need an associated implementation file. You will import it into every code file that contains AppleScript methods you want to be able to debug. You know the drill: Select the Classes group in the Groups & Files pane, use the contextual menu to choose Add > New Group, name the group **Defines**, select it and use the contextual menu again to choose Add > New File, select Other in the source list, select Empty File in the pane to the right, click Next, name the file **AppleScriptLog.h**, add it to the Vermont Recipes and Vermont Recipes SL targets, and click Finish.

Enter the following macro in the new AppleScriptLog.h file:

```
#define    scriptLoggingMasterSwitch    ( 1 )

#if scriptLoggingMasterSwitch
#define SLOG(format,...) NSLog( @"SLOG: File=%s line=%d proc=%s " format,
    strrchr("/") __FILE__, '/') +1, __LINE__, __PRETTY_FUNCTION__,
    ## __VA_ARGS__ )
#else
#define SLOG(format,...)
#endif
```

Don't forget to import the new file into the VRApplicationController+VRApplicationAdditions.m category implementation file. Add this at the top, right after the existing import directive:

```
#import "AppleScriptLog.h"
```

Then add this line at the top of the category implementation file's `-terminologyVersion` method:

```
SLOG(@"terminologyVersion: %@", [NSNumber
    numberWithInt:APPLESCRIPT_TERMINOLOGY_VERSION]);
```

Now, when you build and run the application and run a script accessing the `terminology version` property, you will see a message like this in the Debugger Console: “2009-12-23 08:44:03.110 Vermont Recipes[31450:a0f] SLOG: File=VRApplicationController+VRAppleScriptAdditions.m line=19 proc=-[VRApplicationController(VRAppleScriptAdditions) terminologyVersion] terminologyVersion: 1.” Among other things, this tells you that the script did in fact find and execute your `-terminologyVersion` method. Once you have implemented more complex AppleScript support, similar SLOG messages will tell you what other methods the script also executed, and in what order, and with what results. Errors may be explained, and in the worst case, there will be no SLOG message and you will know that Cocoa Scripting could not find your method at all.

I see one problem with the macro as implemented in Apple's SimpleScripting series: the SLOG messages appear when you run the release version of Vermont Recipes. I believe it is desirable to turn off SLOG messages in the release version of an application so that curious users are not bothered by the vast amount of information this generates in Console.app. To turn off the messages, first delete the `#define` directive at the top of the AppleScriptLog.h header file, leaving the rest

of the directives intact. Instead of defining it with a value of 1 (true) in the header file, as Apple instructs, define it with a setting of 1 in the GCC_PREPROCESSOR_DEFINITIONS_NOT_USED_IN_PRECOMPS (Preprocessor Macros Not Used In Precompiled Headers) field in the “GCC 4.2 – Preprocessing” section of the Build pane of the project’s Info window for the Debug configuration, like this:

```
scriptLoggingMasterSwitch=1
```

Simply leave this out of the build settings in the Release configuration, and Xcode will ignore it when you build the Release version of the application.

There is a more advanced debugging technique that I will leave you to find in the documentation. It involves turning on AppleScript debugging support at the system level.

10. Finally, add additional documentation to the application’s terminology dictionary. Because AppleScript is an extensible language, scripters need your help to understand how to script your application. You should supplement the descriptions of classes, commands, and other elements in the sdef file with some additional HTML documentation.

The ability to add HTML documentation to terminology dictionaries is a relatively new addition to Cocoa Scripting. I was the first third-party developer to release a scriptable application that includes HTML documentation, in Prefab UI Actions, available in a 30-day free trial version at <http://prefabsoftware.com/uiactions/>. Relatively few applications include it at this time, but it is so easy to add and so valuable that every scriptable application should include it.

You’ve already seen what a terminology dictionary looks like to a scripter, in Figure 12.1. You add the text and the Internet link that appear at the top of the Vermont Recipes Suite in Figure 12.1 in this step.

First, however, look at the portion of the dictionary that shows the new **terminology version** property in the Vermont Recipes Suite (**Figure 12.3**). I’ve rearranged the dictionary viewer window and focused on just this one property of the application class. All of the textual information you see was supplied by the basic content of the sdef file, including the **description** attributes that you added to the sdef file.



FIGURE 12.3
The default view of the Vermont Recipes Suite’s application class.

Contrast that with the display of the Standard Suite's application class, supplied entirely by Apple (**Figure 12.4**). You see the application's elements, the application properties supported by the Standard Suite, and the commands it supports. Apple supplied all of this information, including the descriptions, in the Standard Suite.

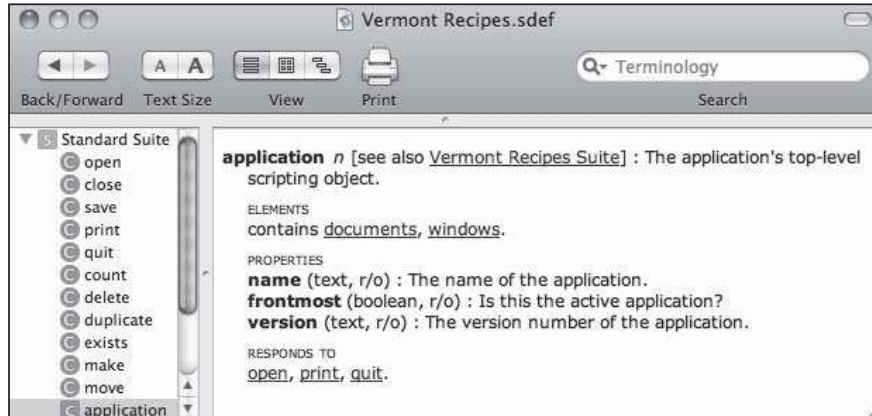


FIGURE 12.4
The default
view of the
Standard Suite's
application class.

Those figures were made with AppleScript Editor's preferences set to their default values. Now open AppleScript Editor's General preferences pane and select the "Show inherited items in dictionary viewer" checkbox. Close and reopen AppleScript Editor's dictionary viewer, and look again at the Standard Suite's application class (**Figure 12.5**). Now you see both the Standard Suite's version and the Vermont Recipes Suite's version of the application class together in one place (I've left out the list of commands). That's very helpful to a scripter who wonders what a script can do with the application class, because there is no longer any need to hunt for inherited versions of the application class in multiple suites throughout the dictionary.

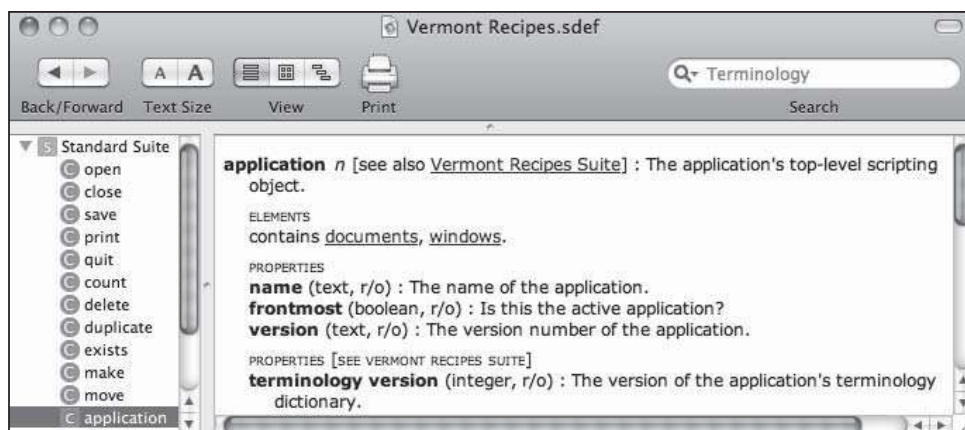


FIGURE 12.5 The combined view of the application class.

Finally, here's what the same dictionary view of the Standard Suite's application class looks like after you add HTML documentation (**Figure 12.6**). It not only provides greater detail regarding the purpose and use of the `terminology version` property, but also even includes an example script.

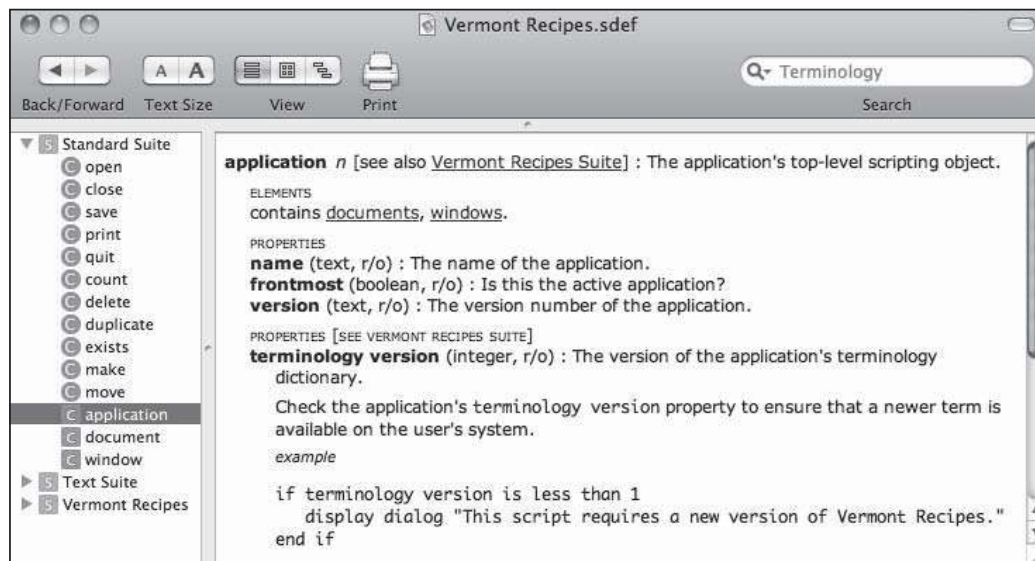


FIGURE 12.6 The same view with additional HTML documentation.

To add the HTML documentation, insert this `documentation` element at the end of the `terminology version` property element in the `sdef` file:

```
<documentation>
  <html><![CDATA[
    <p>Check the application's <code>terminology version</code>
      property to ensure that a newer term is available on the
      user's system.</p>
    <p class="label">example</p>
  <pre><code>if terminology version is less than 1
display dialog "This script requires a new version of Vermont Recipes."
end if</code></pre>
  ]]></html>
</documentation>
```

The code example in the `pre` element is flush left in the `sdef` file to ensure that it will be positioned correctly in the dictionary viewer.

Every `documentation` element contains one or more `html` elements. Place HTML markup tags inside a `CDATA` block or escape them with character entities so that the XML parser won't interpret them as XML.

The `label` class used with the example script is not documented, but I have it on good authority that it is a supported element.

While you're at it, add this **documentation** element at the top of the Vermont Recipes Suite in the **sdef** file, to add the text and the Internet link shown in Figure 12.1:

```
<documentation>
  <html><![CDATA[
    <p>Use the classes and commands in the Vermont Recipes Suite to
      control the Vermont Recipes application and its documents.
      For more information, visit the <a href="http://
        www.vermontrecipes.com">Vermont Recipes Web site</a>.</p>
  ]]></html>
</documentation>
```

Step 3: Add a Diary Document Class and a Property in the Application to Access It

In this step, you add a full-fledged class to the Vermont Recipes Suite, the **diary document** class, which inherits from the Standard Suite's **document** class. The new **diary document** class will form the foundation for adding comprehensive scripting support for a new diary entry class that you will add in Step 5. You also add a **current diary document** property to the extended **application** class in this step to give scripts direct access to the application's current diary document.

The diary document is a one-of-a-kind object—there can be only one diary document in the application at any time. This is known as a *to-one* relationship. In AppleScript, you generally provide access to a to-one relationship by adding a property to its container, as you do in this step by adding the **current diary document** property to the **application** class.

The new **diary entry** class you will create in Step 5 is different in that the diary document can contain many diary entry objects. This is known as a *to-many* relationship. In AppleScript, you generally access objects in a to-many relationship by adding an **element** element to the container. In Step 5, you will add a **diary entries** element to the **diary document** class and support it as an array in your code.

The rules aren't quite as simple as this discussion suggests. You will learn in Step 6 that it is sometimes appropriate to create a property that returns a list of objects. In such

a case, the property is thought of as establishing a to-one relationship with the list, which is of course also a to-many relationship with the items that the list contains.

With the theory out of the way, you are ready to create the **diary document** class and, in the application class, a **current diary document** property that establishes a to-one relationship with a diary document object.

1. Add a new **class** element to the Vermont Recipes Suite, immediately following the **application class-extension** element—that is, after the `</class-extension>` closing tag. The new class is named the **diary document** class, and it looks like this:

```
<class name="diary document" code="dDoc"
      description="The Chef's Diary document." inherits="document">
  <cocoa class="DiaryDocument"/>
</class>
```

In another application, you might have used a **class-extension** element, as you did for the **application** class, if the application supported only one kind of document. Vermont Recipes supports two types of document, a recipes document and a diary document. You haven't done much with the recipes document yet, but you know that the diary document has its own Objective-C subclass, `DiaryDocument`, to implement its features. It is therefore prudent to give the diary document class **name** and **code** attributes of its own, and you specify the `DiaryDocument` class in its **cocoa** element. You also include an **inherits** attribute to indicate that it should pick up all the elements of the Standard Suite's **document** class.

2. While you're working on the `sdef` file, add a **current diary document** property to the **application class-extension** element. This is convenient because scripts can get the diary document without knowing its name.

Add this property element after the **terminology version** property in the **application class-extension** element:

```
<property name="current diary document" code="VRcd"
          type="diary document" access="r"
          description="The current Chef's Diary document, if it is open.">
  <cocoa key="currentDiaryDocument"/>
  <synonym name="current diary" hidden="yes"/>
  <synonym name="diary" hidden="yes"/>
</property>
```

You set the value of its **type** attribute to **diary document**. This is a cross reference to the **diary document** class. It tells Cocoa Scripting that the value of the **current diary document** property is an object having the Objective-C class that was specified in the **cocoa** element of the **diary document** class that you just created, namely, `DiaryDocument`.

You set the **key** attribute of the **cocoa** subelement to **currentDiaryDocument**, which matches the name of the Objective-C method you will write shortly to return the current diary document.

The **synonym** subelements are interesting. Most kinds of elements may have synonyms, which a scripter can enter in place of the element's **name** attribute while writing a script. As soon as the scripter compiles the script, AppleScript Editor changes the script so that it contains the value of the **name** attribute instead. If you are a scripter yourself, you may regularly take advantage of the synonym feature by typing **tell app "TextEdit"** as shorthand for **tell application "TextEdit"** and watching AppleScript Editor automatically convert the shorthand form to the longer official name automatically. The synonyms here—**current diary** and **diary**—are shorter and easier to type. They are marked with the **hidden** attribute in order to avoid cluttering up the dictionary, but you should mention them in any more comprehensive AppleScript documentation you choose to distribute with your application so that users can learn of their availability.

Synonyms are also useful in a more technical sense. For example, if you are upgrading an application, you can provide a new name for an existing element while preserving backward compatibility for older scripts by specifying the old name as a synonym. Old scripts still work, but you discourage new users from using the old name—now a synonym—by marking the synonym with the **hidden** attribute.

3. Now write the code needed to return the new property. This isn't very different from the code you wrote for the **terminology version** property in Step 2, so I'll let you look it up in the downloadable project file for Recipe 12. Treat it as a challenge if you like, and try to write it yourself. Here are some hints: Write the declaration and definition of a **-currentDiaryDocument** accessor method in the **VRApplicationController+VRAppleScriptAdditions** category. This will require you to add a **@class** directive in the header and three **#import** directives in the implementation file.

Finally, don't forget that you must add the key for this property, **currentDiaryDocument**, to the **-application:delegateHandlesKey:** delegate method at the end of the **VRApplicationController.m** implementation file. If you don't do this, a script using the **current diary document** property simply won't work. As originally written, it is a little awkward to add additional keys to the delegate method. Revise it to this new form now:

```
- (BOOL)application:(NSApplication *)sender
    delegateHandlesKey:(NSString *)key {
    if ([[NSArray arrayWithObjects:@"terminologyVersion",
        @"currentDiaryDocument", nil] containsObject:key]) {
        return YES;
    }
    return NO;
}
```

Now you can add an additional key to the comma-separated list any time you add a new property to the application object.

4. The final task in creating the `diary document` class is to write an `-objectSpecifier` method for it. This is not the place to discuss the technical underpinnings of Apple events and AppleScript in Mac OS X, because Cocoa Scripting makes it unnecessary for most developers to know anything about this highly abstruse topic. Just take it on faith that AppleScript objects are represented internally by `NSScriptObjectSpecifier` objects. This method lets Cocoa Scripting know about the containment hierarchy of the class.

Cocoa makes several subclasses of `NSScriptObjectSpecifier` available to you for this purpose. There is an `NSNameSpecifier` class, an `NSIndexSpecifier` class, and several others. You choose whichever one of them returns a reference to the object that you believe will be useful to scripters based on the nature of the object. Most classes use a name specifier, which results in a return value in a script something like this: `document "Chef's diary" of application "Vermont Recipes"`. An index specifier is also commonly used. It returns a value in a script something like this: `diary entry 1 of document "Chef's Diary" of application "Vermont Recipes"`.

Write the `-objectSpecifier` method now. Because it specifies a diary document, it belongs in the existing `DiaryDocument` class. However, you have adopted the approach of placing AppleScript support in separate files, so you must now create a pair of `DiaryDocument+VRAppleScriptAdditions` files to hold a new category on `DiaryDocument`. The process is almost identical to that which you followed in Step 2 to create the `VRApplicationController+VRAppleScriptAdditions` category.

Select the AppleScript Support group in the Xcode project window's Groups & Files pane. Using the contextual menu, choose `Add > New File` and create another pair of Cocoa source files. In the New File dialog, select Cocoa Class in the source list on the left, select "Objective-C class" in the upper-right pane, choose subclass of `NSObject` in the pop-up menu below that, click Next, name the file **`DiaryDocument+VRAppleScriptAdditions.m`**, select the checkbox to create a header file with the same name, select both the Vermont Recipes and Vermont Recipes SL targets, click Finish, and add your customary information to the top of both files.

Add the category code. In the new category header file, import the base class's header file by adding an import directive to the list of imported files, like this:

```
#import "DiaryDocument.h"
```

Change the `@interface` directive to this, without any curly braces:

```
@interface DiaryDocument (VRAppleScriptAdditions)
```

Also change the `@implementation` directive in the category implementation file to this:

```
@implementation DiaryDocument (VRAppleScriptAdditions)
```

Finally, write the `-objectSpecifier` method at the end of the implementation file:

```
#pragma mark APPLESCRIPT SUPPORT
```

```
- (NSScriptObjectSpecifier *)objectSpecifier {
    NSScriptObjectSpecifier *specifier = [[NSNameSpecifier alloc]
        initWithContainerClassDescription:[NSScriptClassDescription
            classDescriptionForClass:[NSApp class]] containerSpecifier:nil
        key:@"orderedDocuments" name:[self lastComponentOfFileName]];
    SLOG(@"Diary Document objectSpecifier: %@", specifier);
    return [specifier autorelease];
}
```

This does nothing very complicated. It creates and returns an instance of the `NSNameSpecifier` class, initializing it with a container class description and some other information. Remember that the role of the `-objectSpecifier` method is to inform Cocoa Scripting about the containment hierarchy of the object. When the application object is the container, special rules apply. You'll learn about writing an `-objectSpecifier` method for a class deeper in the containment hierarchy later.

When the application is the container, you get the required class description by calling the `+classDescriptionForClass:` class method on the `NSApplication` class. The `containerSpecifier` parameter value is `nil` in the special case where the application is the container. It must be an actual object specifier for any other container. You set the key for the diary document to `@orderedDocuments` because the application object holds all of its open documents in an array returned by the `-orderedDocuments` method. Finally, you set the name of the document for the name specifier by calling `NSDocument`'s `-lastComponentOfFileName` method. All of `NSDocument`'s methods are available, as well as `DiaryDocument`'s methods, because this is a category on `DiaryDocument`, which is a subclass of `NSDocument`.

Because you use the `SLOG` macro, you must also import the `AppleScriptLog.h` header file. Add this directive to the imports list in the `DiaryDocument+VRAppleScriptAdditions.m` implementation file:

```
#import "AppleScriptLog.h"
```

Step 4: Add the Text Suite and a Document Text Property

The diary document contains nothing but rich text. It is not a database. Although it is organized conceptually into separate diary entries having a defined structure, it has no inherent structure. It's just a stream of rich text.

This has two consequences for purposes of AppleScript support. First, a scripter will find it very useful to be able to manipulate the text using the well-defined and established text-handling features of the AppleScript Text Suite, which supports reading and writing text broken into characters, words, sentences, and paragraphs. Second, a scripter will want to manipulate the several conceptually separate diary entries as if they were records in an array.

AppleScript and Cocoa Scripting are adept at achieving both goals. Cocoa Scripting is designed so that you can easily manipulate any data as if it were an array of ordered items or a set of unordered items, even if the data is not organized in that way.

In this step, you implement the first goal, adding the ability to script the diary document as text. In Step 5, you will create a new Objective-C class, `DiaryEntry`, strictly for the purpose of allowing AppleScript to treat the diary document as an ordered array of separate diary entry records having defined fields. The `DiaryEntry` class will support the new **diary entry** class in the `sdef` file.

1. The Text Suite is another terminology suite provided by Apple, like the Standard Suite. Currently, the Text Suite differs from the Standard Suite in that a definitive version is not in the `/System/Library/ScriptingDefinitions` folder where it could be included in application dictionaries using the `XInclude` mechanism. Instead, you must find it somewhere else and paste it into your `sdef` file.

The first promising source you might stumble upon is the `Sketch.sdef` file in the Sketch example project in your Developer folder at `/Developer/Examples/Sketch`. However, Apple has unofficially acknowledged that this version of the Sketch.sdef file contains an error. It defines the **rich text** class using `ctxt` as its code. That code is also used by AppleScript to define the **text** class, and the resulting terminology conflict has led to problems. In the *Cocoa Scripting Guide*, Apple recommends that you instead use `ricT` as the code for the **rich text** class.

Until Apple places a definitive Text Suite in the `/System/Library/ScriptingDefinitions` folder, I recommend that you use the most recent complete version of the Text Suite, which appears in Apple's ScriptingDefinitions 1.3 sample code. It is available through the Xcode documentation window. It uses the correct `ricT` code for the **rich text** class, and it uses `TEXT` as the code for the Text Suite instead of ????.

Open the Sketch.sdef file in the ScriptingDefinitions 1.3 example code, copy the `Text Suite` element, and paste it into the `VermontRecipes.sdef` file right after the `Standard Suite XInclude` element. Be careful not to include the XML header when you copy the `Text Suite`, but only the suite itself.

2. Now add a `document text` property to the `diary document` class. Scripters can use this to write a script to get the entire text of the diary document like this: `get document text of current diary document`. Then they will be able to manipulate the text in the result using the properties of the `Text Suite`, like this: `get paragraph 1 of result`.

Start by adding this element to the `diary document` class in the `VermontRecipes.sdef` file, after the `cocoa` element:

```
<contents name="document text" code="VRft" type="rich text"
    description="The full text of the document.">
    <cocoa key="documentText"/>
</contents>
```

Notice that this is a special kind of `property` element, a `contents` element. A `contents` element acts in every respect like a `property` element, except that it has an additional effect. As explained in the `sdef(5)` man page, it makes the `document text` property an *implied container*. This means that scripters can, at their option, omit it from the containment hierarchy when referring to properties of the `document text` property. An example makes this clear. If the `document text` property were defined using a `property` element, a script to get its first word would have to be written like this: `get word 1 of document text of current diary document`. When a `contents` element is used instead, the same script can be written, at the scripter's option, in this shortened form: `get word 1 of current diary document`.

Only one property per class can be a `contents` element. It is therefore important to consider which of a class's properties, if any, is the one best designated as an implied container. As with `document text`, it should be a property that can appropriately stand in for the object itself.

Another point to notice about this property is that it contains no `access` attribute. Since the default value is `rw` for read/write, this means that scripts will be able to set the value of the property, not just read its value.

3. Write the getter accessor method for the `document text` property. Since it is a read-write property, you will eventually write a setter accessor as well, but I will defer setters to Step 6 because they are more complex. For now, in the `Diary Document+VRAppleScriptAdditions.h` header file, declare the getter accessor method like this:

```
#pragma mark ACCESSOR METHODS

- (NSTextStorage *)documentText;
```

In the implementation file, define the getter like this:

```
#pragma mark ACCESSOR METHODS
```

```
- (NSTextStorage *)documentText {
    NSTextStorage *storage = [[NSTextStorage alloc] init];
    [storage setAttributedString:[self diaryDocTextStorage]];
    SLOG(@"documentText: %@", storage);
    return [storage autorelease];
}
```

The getter method is relatively straightforward. It creates and initializes an empty `NSTextStorage` object, sets its attributed string to the value returned by the `DiaryDocument` class's `-diaryDocTextStorage` method, and returns it. You might be tempted to return a copy of the document's existing text storage object instead of creating a new object, but don't do it. I find that this only works if you create a new text storage object. I assume this has something to do with the additional information that the document's text storage object has acquired.

Returning an object of type `NSTextStorage` is a requirement for all **rich text** classes. `NSTextStorage` implements several methods specifically designed to support the properties of the Text Suite, including a `-characters` method that returns an array of the characters in the text object much more efficiently than you could do by attempting to create this functionality yourself.

4. The ability to treat the `document text` property as an implied container is not obvious from the brief description in the dictionary. This is a good candidate for some additional HTML documentation. Add this **documentation** element at the end of the `document text` property in the `sdef` file, after the `cocoa` subelement:

```
<documentation>
  <html><![CDATA[
    <p>The <code>document text</code> property is optional when
    referring to Text Suite objects (such as <code>words</code> and
    <code>paragraphs</code>) in the application's <code>current diary
    document</code>.</p>
    <p class="label">examples</p>
    <pre><code>get paragraph 3 of current diary document
get paragraph 3 of document text of current diary document</code></pre>
  ]]></html>
</documentation>
```

Step 5: Add a Diary Entry Class and an Element in the Diary Document to Access It

You learned in Steps 2, 3, and 4 about to-one relationships in AppleScript, which are known as *properties*. In this step, you create a to-many relationship, which is known in AppleScript as an *element*. You add to the **diary document** class a new **diary entries** element, and you create a new **diary entry** class in the `sdef` file so you can fill the element with multiple diary entry objects.

Be careful as you read this step to distinguish between the three senses of the term *element*. I sometimes refer to an *AppleScript element*—an AppleScript to-many relationship; I sometimes refer to an *XML element* in the `sdef` file—an XML syntax feature; and I sometimes refer to an *element XML element* in the `sdef` file—a specific kind of XML element that corresponds to an AppleScript element.

This is the beginning of a series of steps relating to the **diary entry** class. In these steps, you will learn how to set the values of properties instead of just getting them, about getting items from AppleScript elements and adding items to them, and about various kinds of AppleScript commands. In this step, you start by setting up an AppleScript element and getting individual items from it.

1. In the `sdef` file, add a `class` element for the new **diary entry** class. Place this element after the **diary document** `class` element:

```
<class name="diary entry" code="dEnt"
    description="An entry in the Chef's Diary."
    plural="diary entries" inherits="item">
  <cocoa class="DiaryEntry"/>
  <synonym name="entry" hidden="yes"/>
</class>
```

By now you may have noticed that you don't nest `class` elements in the `sdef` file. Instead, the containment hierarchy of classes in the application is dictated by the `-objectSpecifier` methods you implement for each class and by the properties and relationships in the `sdef` file.

The only attribute that is new to you here is the `plural` attribute. You should use it in any element whose name has an irregular plural. Without a `plural` attribute, AppleScript would generate the incorrect term **diary entrys** by adding “s” to the singular name. AppleScript automatically use the plural form of a class's name when referring to an element containing multiple instances of its items, whether the plural is generated automatically or by a `plural` attribute.

You include an `inherits` attribute for the built-in AppleScript `item` class only because this causes the human-readable dictionary to display this inheritance relationship. Even without the `inherits` attribute, the class will respond to properties of the `item` class, such as the `class` property and the `properties` property.

The Cocoa class that supports the `diary entry` class is `DiaryEntry`, which you will write shortly.

2. Still in the `sdef` file, add this element to the `diary document` class after the `document text contents` element:

```
<element type="diary entry">
  <cocoa key="orderedEntries"/>
</element>
```

The `cocoa` subelement's `key` attribute is `orderedEntries`, identifying the `-orderedEntries` accessor method that you will write shortly. The Vermont Recipes human-readable dictionary will present the element using its plural form, *diary entries*.

3. Before you create the new `DiaryEntry` class, consider the code that you must write to support the `diary entries` element. Vermont Recipes implements the element in code as an array, and the code accesses the array using the `orderedEntries` key. You must attend to several details, including writing the `-orderedEntries` getter accessor method to get the list of diary entries. You can write a standard setter method, too, but that generally is the least efficient way to modify an AppleScript element. Instead of a setter accessor method, you use key-value coding (KVC) methods to insert, remove, and replace individual items in the list. You also support the `make` command to create a new object and, in the process, insert it into an existing element at an appropriate location. You deal with the getter side of the implementation in this step; you will deal with making new diary element objects and modifying the array in Step 8.

Although for efficiency reasons you should not ordinarily use a standard setter accessor method to modify an AppleScript element, you will need to write a setter for the `orderedEntries` object in this step. It is required by the internal implementation details of the `diary entries` element. Specifically, you will have to invalidate the `orderedEntries` array periodically and then recreate it lazily when a script is run, because the array must be revised when the user edits the diary document. The setter is for internal use by your code when this happens, not to support an AppleScript `set` command. If you implement so-called indexed accessor methods to modify an AppleScript element, as you will do in Step 8, Cocoa Scripting automatically uses these more efficient indexed accessor methods in preference to the setter accessor when a script modifies the element.

In a typical scriptable application, each item in the array would be an object that contains all of the data needed by a script to access the object's properties. In Vermont Recipes, however, the individual diary entry objects do not contain any text at all. The text is already available in the DiaryDocument's **diaryDocTextStorage** instance variable, and it would be wasteful to duplicate it in the individual diary entries. Instead, each diary entry is a very small object holding only an NSRange structure that records the start location and length of its text in the diary document's text storage object. To get the text of a diary entry, you extract the text from the given range in the text storage object.

The ranges of the diary entries are volatile because editing the text of any diary entry alters the ranges of every diary entry located after it in the array. To deal with this, you must update the ranges in the diary entry objects before running a script if the diary document has been edited since the last script was run. You invalidate the existing array when the user edits the document, and you recreate it when a script is run if the document has been edited. You invalidate the array by setting it to **nil**, and if it is **nil**, you recreate it with new diary entry objects containing the newly revised ranges. You will write a **-makeOrderedEntries** method to create and recreate the array, an **-invalidateOrderedEntries** method to invalidate it, and a **-setOrderedEntries:** accessor method to install it every time it is created or recreated. The array and these three methods support to-many access to the **diary entries** AppleScript element.

The first aspect you address in this step is the mechanism for invalidating the array of diary entry objects every time the diary document's content changes. NSTextStorage declares a **-textStorageDidProcessEditing:** delegate method for responding to changes made to text. TextEdit uses it in its AppleScript implementation, and you will use it here.

You can't use a delegate method without first setting up the delegate connection, so do that now. It is the diary document's text storage object that needs a delegate to help it out, and the diary document should be its delegate. Add this simple statement at the end of the **-setDiaryDocTextStorage:** method in the DiaryDocument.m implementation file:

```
[diaryDocTextStorage setDelegate:self];
```

The **-setDiaryDocTextStorage:** method is called every time the diary document's window is opened and when the diary document is read from disk.

This requires declaring that DiaryDocument implements the NSTextStorageDelegate protocol, so revise the **@interface** directive in the DiaryDocument.h header file to this:

```
@interface DiaryDocument : NSDocument <NSTextStorageDelegate>
```

Leave the implementation of the `-textStorageDidProcessEditing:` delegate method until later. First, write the code needed to support the array.

4. Declare an `orderedEntries` instance variable and write its getter and setter accessor methods. The `orderedEntries` instance variable and its accessors are named by analogy to `NSApplication`'s `-orderedDocuments` and `-orderedWindows` methods. The name `orderedEntries` is specified in the sdf file's `diary` `entries` element as its `cocoa` element's `key` attribute. Since Vermont Recipes needs this array only for AppleScript support, its accessors belong in the `DiaryDocument+VRApplScriptAdditions` category.

You can't declare instance variables in category files, however, so declare the `orderedEntries` instance variable in the `DiaryDocument.h` header file. Add this at the end of the instance variables:

```
NSMutableArray *orderedEntries;
```

The `orderedEntries` instance variable is an `NSMutableArray` object. Scripts can add objects to it and remove objects from it at any time.

It will be created and initialized lazily whenever a script is run after the user has edited the document, so you should not initialize it in the `DiaryEntry`'s initialization methods. You don't have to take special steps to set it to `nil` when the document is created, either, because instance variables representing objects are set to `nil` automatically. Nevertheless, it still has to be deallocated when the diary document is closed. Add a `-dealloc` method just after the `-init` method, like this:

```
- (void)dealloc {
    [orderedEntries release];
    [super dealloc];
}
```

Now declare the setter and getter at the top of the Accessor Methods section of the `DiaryDocument+VRApplScriptAdditions.h` category header file like this:

```
- (void)setOrderedEntries:(NSMutableArray *)array;
- (NSMutableArray *)orderedEntries;
```

Define them like this:

```
- (void)setOrderedEntries:(NSMutableArray *)array {
    if (orderedEntries != array) {
        [orderedEntries release];
        orderedEntries = [array retain];
    }
}
```



```

- (NSMutableArray *)orderedEntries {
    NSLog(@"orderedEntries: %@", orderedEntries);
    if (!orderedEntries)
        [self setOrderedEntries:[self makeOrderedEntries]];
    return [[orderedEntries retain] autorelease];
}

```

The setter is written like any other setter accessor method. This would be sufficient to support scripts that modify the `orderedEntries` element if you did not implement indexed accessor methods. However, because it won't be as efficient as the indexed accessor methods, you will use the indexed accessor methods to modify the `orderedEntries` element in Vermont Recipes when you get to Step 8.

The getter returns the value of its instance variable like any other getter accessor method. First, however, unlike a standard getter accessor, it checks to make sure the array exists. The array will be `nil` if the diary document has just been opened or if the document has been edited since the last script was run, marking it as invalid. If it is `nil`, the getter does not return `nil` as a standard getter would, but it instead creates a new array whose diary entries reflect the revised text ranges in the document. Then the getter installs the array by calling the setter, before returning the new array to the caller. By installing the new array, the getter marks it as valid so that the next script can use it without recreating it—if the user hasn't edited the document and invalidated the array again in the meantime.

Pause now for a moment to consider the reasons to use an array and accessor methods to implement the `diary entries` element. It is often natural and convenient to maintain a list of application data in the form of an array, for reasons unrelated to AppleScript. When you do create an array and write an accessor method, Cocoa Scripting automatically uses them when a script accesses items in the list, by taking advantage of key-value coding.

But it is often awkward to use an array to maintain an application's data as a list of objects. The diary document is an example of data that does not intrinsically take the form of an array but which can nevertheless be thought of as a list of objects. The document's content is a single stream of text, and the idea that the text is broken into individual diary entries is purely conceptual. To get the text of any one diary entry, the application must parse the stream of text looking for the entry marker characters that define the beginning of each diary entry. In Vermont Recipes, it is possible to create an array to support the concept, but it won't necessarily be easy or efficient to do that in another application.

For cases like this, Cocoa Scripting provides an alternative technique that you can use to make AppleScript think the data is a list of objects even if it isn't in an array. Instead of parsing the text to create an array, as you do in Vermont

Recipes, you could leave the data in the form of a text stream and parse it anew every time a script needs to get a diary entry. You wouldn't need the array at all.

When you use this alternative approach, you don't write an accessor method, because KVC can only use an accessor method for an AppleScript element if the data is held in an array. Instead, you write two KVC methods, `-countOf<Key>` and `-objectIn<Key>AtIndex:`. These are known as *indexed accessors*. In the case of the diary entries element, you would name these `-countOfOrderedEntries` and `-objectInOrderedEntriesAtIndex:`, substituting `OrderedEntries` for `<Key>`. You would write the `-countOfOrderedEntries` method so that it scans the text in the diary document and returns the total number of paragraphs that begin with an entry marker character. You would write the `-objectInOrderedEntriesAtIndex:` method so that it does the same thing, except that it would stop scanning the text when it found the diary marker at the specified index. It would then create and return a new diary entry object holding the range of that diary entry. The code to scan the text might be quite similar to the `-makeOrderedEntries` method that you will write shortly to create the array that Vermont Recipes uses.

I won't reproduce here the specific code these indexed accessor methods might use. However, the downloadable project files for the book include a version of the finished application in which indexed accessor methods take the place of the array and the `-orderedEntries` and `-setOrderedEntries:` accessor methods. It is named *Vermont Recipes 2.0.0 - Recipe 14 Alternate*. To find all of the code changes in it that relate to this alternative way to support AppleScript, search the project for the phrase *RECIPE 14 ALTERNATE*. The project also includes an alternate technique for inserting and removing diary entries without using an array, discussed in Step 8.

In the end, whether you choose to use an array with a traditional getter accessor method or, instead, the indexed accessor methods without an array is largely a matter of performance. Both approaches require about the same amount of code because, either way, you need a method to parse the text looking for entry marker characters. In my judgment—which is not based on systematic testing—the use of the array with a getter accessor method is probably more efficient in Vermont Recipes, because many scripts can be run without recreating the array as long as the user does not edit the text, and the NSArray methods that Cocoa Scripting uses internally to extract indexed items are more efficient than parsing the text repeatedly.

5. With that design decision behind you, you can now create the `orderedEntries` array. To do this, write a `-makeOrderedEntries` method that is called from the `-orderedEntries` accessor method. The method creates new diary entry objects by parsing the current diary document's textual content from beginning to end. It returns the finished array to the caller. The completed array is empty if there are no diary entries in the document.

Declare the method at the end of the `DiaryDocument+ VRApplScriptAdditions.h` header file like this:

```
#pragma mark APPLESCRIPT SUPPORT
```

```
- (void)makeOrderedEntries;
```

Define it in the `DiaryDocument+ VRApplScriptAdditions.m` implementation file like this:

```
#pragma mark APPLESCRIPT SUPPORT
```

```
- (NSMutableArray *)makeOrderedEntries {
    NSMutableArray *array = [NSMutableArray array];
    NSRange thisEntryRange = [self firstEntryTitleRange];
    if (thisEntryRange.location != NSNotFound) {
        DiaryEntry *entry;
        NSRange nextEntryRange =
            [self nextEntryTitleRangeForIndex:
             thisEntryRange.location + thisEntryRange.length];
        while (nextEntryRange.location != NSNotFound) {
            thisEntryRange.length =
                nextEntryRange.location - thisEntryRange.location;
            entry = [[DiaryEntry alloc] initWithRange:thisEntryRange
                forDocument:self];
            [array addObject:entry];
            [entry release];
            thisEntryRange = nextEntryRange;
            nextEntryRange = [self nextEntryTitleRangeForIndex:
                thisEntryRange.location + thisEntryRange.length];
        }
        thisEntryRange.length = [[self diaryDocTextStorage] length]
            - thisEntryRange.location;
        entry = [[DiaryEntry alloc] initWithRange:
            thisEntryRange forDocument:self];
        [array addObject:entry];
        [entry release];
    }
    return array;
}
```

In previous recipes, you have written a lot of code using the methods in the `DiaryDocument` class to manipulate diary entries, so this shouldn't require much explanation. The `-makeOrderedEntries` method makes a single pass through the document's text storage object, calculating the range of each diary

entry it encounters. It creates a new diary entry object for each entry and initializes it with the range of the entry. It also passes a reference to `self` so that the diary entry object can refer to the diary document when it needs to get the text in the specified range. It creates each diary entry object by calling the diary entry's `-initWithRange:forDocument:` method, which you will write shortly in the new `DiaryEntry` class.

Don't forget to import the `DiaryEntry.h` header file, which you haven't yet created. Add this at the end of the imports list at the top of the `DiaryDocument+VRAppleScriptAdditions.m` implementation file:

```
#import "DiaryEntry.h"
```

6. You haven't yet invalidated the `orderedEntries` array when the diary document's content is edited. To make the code more understandable, write a separate method, `-invalidateOrderedEntries`, to do the job. In the `DiaryDocument+VRAppleScriptAdditions.h` header file, declare it like this following the new `-makeOrderedEntries` method:

```
- (void)invalidateOrderedEntries;
```

Define it like this in the implementation file:

```
- (void)invalidateOrderedEntries { // ADDED IN RECIPE 12
    [self setOrderedEntries:nil];
}
```

In addition to calling the `-invalidateOrderedEntries` method whenever the user edits the diary document, you must call it in the `DiaryDocument`'s `-setDiaryDocTextStorage` method. Add this statement at the end of the `-setDiaryDocTextStorage` method:

```
[self invalidateOrderedEntries];
```

This requires importing the category's header file, so add this to the imports list at the top of the `DiaryDocument.m` implementation file:

```
#import "DiaryDocument+VRAppleScriptAdditions.h"
```

7. Now you can write the delegate method that invalidates the `orderedEntries` array whenever the document's text contents are changed, whether by manual editing or by running a script. Insert the delegate method at the end of the `DiaryDocument+VRAppleScriptAdditions.m` implementation file:

```
#pragma mark DELEGATE METHODS
```

```
- (void)textStorageDidProcessEditing:(NSNotification *)notification {
    [self invalidateOrderedEntries];
}
```

8. At last, you are ready to create the `DiaryEntry` class. This involves several tasks.

Start by creating the new source files. The `DiaryEntry` class exists only to support AppleScript, so it belongs in the AppleScript Support group. Select that group in the Groups & Files pane in the Xcode project window. Using the contextual menu, choose `File > New File`, select `Cocoa Class` in the source pane, select “Objective-C class” to the right, choose `Subclass of NSObject`, click `Next`, name the file **`DiaryEntry.m`**, select the checkbox to create the header file, select both targets, click `Finish`, and add the usual information at the top of both files.

9. Declare two instance variables in the `DiaryEntry.h` header file like this:

```
NSRange entryRange;
DiaryDocument *diaryDocument;
```

Because the `DiaryDocument` class appears in the header file, add this forward reference after the import list at the top of the header file:

```
@class DiaryDocument;
```

You’ll import `DiaryDocument.h` and other required header files into the `DiaryEntry.m` implementation file at the end of this task.

Declare the class’s basic initialization and accessor methods in the header at the same time:

```
#pragma mark INITIALIZATION
```

```
- (id)initWithRange:(NSRange)range forDocument:(DiaryDocument *)document;
```

```
#pragma mark ACCESSOR METHODS
```

```
- (NSRange)entryRange;
- (DiaryDocument *)diaryDocument;
```

Define all the basic methods in the implementation file like this:

```
#pragma mark INITIALIZATION
```

```
- (id)init {
    return [self initWithRange:NSMakeRange(0, 0) forDocument:nil];
}

- (id)initWithRange:(NSRange)range forDocument:(DiaryDocument *)document {
    if ((self = [super init])) {
        entryRange = range;
        diaryDocument = [document retain];
    }
    return self;
}
```

(code continues on next page)

```

- (void)dealloc {
    [diaryDocument release];
    [super dealloc];
}

#pragma mark ACCESSOR METHODS

- (NSRange)entryRange {
    return entryRange;
}

- (DiaryDocument *)diaryDocument {
    return diaryDocument;
}

```

10. As you know, all classes used by Cocoa Scripting must have an `-objectSpecifier` method to inform Cocoa Scripting of their place in the application's containment hierarchy. Add this method at the end of the `DiaryEntry.m` implementation file:

```

#pragma mark APPLESCRIPT SUPPORT

- (NSScriptObjectSpecifier *)objectSpecifier {
    NSMutableArray *orderedEntries =
        [[self diaryDocument] orderedEntries];
    NSUInteger idx = [orderedEntries indexOfObjectIdenticalTo:self];
    if (idx == NSNotFound) return nil;
    NSScriptObjectSpecifier *containerRef =
        [[self diaryDocument] objectSpecifier];
    NSScriptObjectSpecifier *specifier = [[NSIndexSpecifier alloc]
        initWithContainerClassDescription:[containerRef
        keyClassDescription] containerSpecifier:containerRef
        key:@"orderedEntries" index:idx];
    SLOG(@"Diary Entry objectSpecifier: %@", specifier);
    return [specifier autorelease];
}

```

This is a little different from the `DiaryDocument` class's `-objectSpecifier` method, because the `DiaryEntry` class's container is not the top-level application object but the `DiaryDocument` class. In addition, it returns an index specifier instead of a name specifier.

It first determines the entry's index in the `orderedEntries` array using `NSArray`'s `-indexOfObjectIdenticalTo:` method. Then, if it is found in the array, the

method gets the object specifier of its container by calling `DiaryDocument`'s `-objectSpecifier` method, which you wrote earlier. That method in turn gets the object specifier for the containing application object, which is always `nil`. In this fashion, it returns a specifier describing its entire containment hierarchy. The value of the `key:` parameter is, of course, `@"orderedEntries"` for the `-orderedEntries` accessor method.

11. Import the `DiaryDocument+VRAppleScriptAdditions.h` header file because that's where the `-orderedEntries` method is declared. You also have to import some other headers. Add all of these to the `DiaryEntry.m` implementation file now:

```
#import "AppleScriptLog.h"
#import "DiaryDocument.h"
#import "DiaryDocument+VRAppleScriptAdditions.h"
```

12. That's it for the basic infrastructure of the `DiaryEntry` class and the `diary entries` AppleScript element. A scripter can now access any diary entry in the document by its index using a script statement like `get diary entry 3 of current diary document`.

In Step 6, you will make the new diary entry class useful by implementing several AppleScript properties for it. These will enable scripters to get and set the name, date, tags, body text, and entry text of any diary entry. In Step 7, you will create a `current diary entry` property of the diary document based on the current location of the insertion point. Finally, you will learn about a variety of techniques for creating AppleScript commands in the remaining steps. The first two of them, Steps 8 and 9, will complete the discussion of the `diary entries` element by showing you how to make a new diary entry, to insert it in the `diary entries` element, and to delete it from the `diary entries` element. In those two steps, you will learn about two alternate approaches that you can take to modify the `diary entries` element.

Step 6: Add Properties to Get and Set Diary Entry Values

One of the reasons to create a `diary entry` class is to enable a script to get and set the text of the diary document in a variety of ways. In this step, you implement that capability by adding properties to the `diary entry` class. You also complete a task you started in Step 4 by enabling a scripter to set the full text of the diary document as well as getting it.

1. As usual, start by adding elements to the sdef file. Add these **property** elements to the diary entry class following the **synonym** subelement:

```
<property name="name" code="pnam" type="text"
    description="The title of the entry.">
    <cocoa key="name"/>
    <synonym name="title" hidden="yes"/>
</property>

<property name="date" code="dEdt" type="date"
    description="The date of the entry as shown in its title.">
    <cocoa key="date"/>
</property>

<property name="tags" code="dEts" description="The tags for the entry.">
    <cocoa key="tags"/>
    <type type="text" list="yes"/>
</property>

<property name="body text" code="dEbt" type="rich text"
    description="The body text of the entry, omitting its date and tags.">
    <cocoa key="bodyText"/>
</property>

<contents name="entry text" code="dEft" type="rich text"
    description="The full text of the entry, including
        its date and tags.">
    <cocoa key="entryText"/>
</contents>
```

The **name** property is plain text, not rich text. It is a standard property having an all-lowercase code, **pnam**, provided by Apple. Whenever a standard property serves your purposes, you should use it together with its standard code in order to promote consistency and avoid proliferation of new terminology.

The Vermont Recipes specification contemplates that a diary entry's name is the date when it was created. However, since the diary document is simply a stream of text, the user can violate this rule at will, either by typing a new name or running a script with a **set name** statement. Doing this has adverse consequences. Among other things, the **date** property returns **missing value** if the name cannot be parsed as a valid date. You could deal with this issue by marking the **name** property read-only, but here you give the user the option to provide any value. You are expected to document design decisions like this in the application's documentation when you release it, but it makes sense to warn scripters

in the dictionary as well. Add this `documentation` element to the `name` property element after the `synonym` element:

```
<documentation>
  <html><![CDATA[
    <p>Set the <code>name</code> property to a value representing
      a valid date, such as "January 1, 2010 10:05:49 PM EST";
      otherwise, <code>get date</code> returns <code>missing
      value</code>. It is better to set the name by using
      <code>set date</code>.</p>
  ]]></html>
</documentation>
```

The `date` property holds an AppleScript date value, represented in Cocoa by the `NSDate` class. The `tags` property holds an AppleScript list value, represented in Cocoa by the `NSArray` class. The `body` text property is AppleScript rich text, represented in Cocoa by the `NSTextStorage` class.

The `entry` text property is also rich text. It is a `contents` element, not a `property` element, so it is an implied container just like the diary document's `document` text `contents` element. As you did with the `document` text element, document this in the dictionary with this `documentation` element:

```
<documentation>
  <html><![CDATA[
    <p>The <code>entry text</code> property is optional when
      referring to Text Suite objects (such as
      <code>words</code> and <code>paragraphs</code>) in
      a <code>diary entry</code>.</p>
    <p class="label">examples</p>
    <pre><code>get word 1 of diary entry 2 of current diary document
get word 1 of entry text of diary entry 2 of current diary document
</code></pre>
  ]]></html>
</documentation>
```

2. Declare the getter and setter accessor methods supporting these properties, all of which are read-write properties. Add these declarations to the `DiaryEntry.h` header file at the end of the Accessor Methods section:

- (void)setName:(NSString *)name;
- (NSString *)name;
- (void)setDate:(NSDate *)date;
- (NSDate *)date;

(code continues on next page)

- (void)setTags:(NSArray *)tags;
- (NSArray *)tags;

- (void)setBodyText:(id)text;
- (NSTextStorage *)bodyText;

- (void)setEntryText:(id)text;
- (NSTextStorage *)entryText;

The `-bodyText` and `-entryText` getters return `NSTextStorage` objects because they represent rich text properties based on the Text Suite. The setters take untyped parameters because a script might pass in either rich text or plain text, as discussed in a moment.

3. Define the getter accessor methods at the end of the `DiaryEntry.m` implementation file's Accessor Methods section. I'll get you started by spelling out how to insert the `-name` method here. I'll leave the rest of the getters to you. They are all similar to the `-name` method in that they use methods from the `DiaryDocument` class that you wrote previously to calculate the range of text to be retrieved from the document's text storage object. They are fully implemented in the downloadable project file for Recipe 12.

Here's the `-name` method:

```
- (NSString *)name {
    NSString *entryString = [[[[self diaryDocument] diaryDocTextStorage]
                             string] substringWithRange:[self entryRange]];

    NSString *nameString;
    NSRange nameRange = [DiaryDocument rangeOfLineFromMarkerIndex:0
                             inString:entryString];
    if (nameRange.length <= 2) {
        nameString = (NSString *)[NSNull null];
    } else {
        nameRange.location += 2;
        nameRange.length -= 2;
        nameString = [entryString substringWithRange:nameRange];
        if (nameString == nil) nameString = (NSString *)[NSNull null];
    }
    SLOG(@"name: %@", nameString);
    return nameString;
}
```

This starts by extracting the entry's full text as an `NSString` object from the document's text storage using the `DiaryEntry` object's `-entryRange` accessor

method. It is this technique that you planned to support when you originally decided to store each entry's range in a `DiaryEntry` object, instead of storing a duplicate of the text itself. The other accessor methods will use similar techniques to get at their values.

An `NSString` object is also appropriate here because it makes it easy to calculate the range of the entry's name. You adjust the range to eliminate the leading entry marker and its following space character.

If the entry does not have a name, you return `[NSNull null]`, which starting with Leopard causes AppleScript to report `missing value`. Cocoa's `+[NSNull null]` class method returns a real object, whereas `nil` denotes the absence of an object. The method could have been declared to return `id`, an untyped object, to accommodate the `NSNull` object, which is not an `NSString` object. However, I find the code to be more informative when the expected class of the returned object is specified, and here it is easy to cast the `NSNull` object to `NSString`. AppleScript still sees it as a `missing value`.

4. Next, write the `-date` method yourself as a challenge. Here's a hint: Remember the `DiaryDocument`'s `-dateFromEntryTitleRange:` method, which you wrote long ago to convert a diary entry's name into an `NSDate` object. You'll want to return `[NSNull null]` if the name can't be converted to a valid date.
5. I leave it to you to write the remaining getter accessors. They are fully implemented in the downloadable project file for Recipe 12. Here are some explanations of the code you can find there:

For the `-tags` method, extract each tag as a string from the tags list in the diary document's text storage object; then add it to an array and return the array. Use `NSString`'s handy `-componentsSeparatedByCharactersInSet:` and `-stringByTrimmingCharactersInSet:` methods to assist in these tasks.

In the `-name`, `-date`, and `-tags` getter methods, you are able to do all calculations in the entry string as an `NSString` object, because you are not interested in them as rich text. In `-bodyText`, however, you return an `NSTextStorage` object. You can still calculate the locations in the entry string as before using `NSString`, making sure to adjust for newline characters—one for the name, plus one for the tag list if there is a tag list. You then adjust the ranges by adding the offset of the diary entry in the diary document as a whole. This allows you to extract this entry's text from the full text storage object as rich text. The `-entryText` method is similar.

6. Turn now to the setter methods. You set AppleScript properties using ordinary Cocoa setter accessor methods.

However, the setters are more interesting than the getters. A setter is supposed to change the value of an object in the application—in this case, the current

diary document. Thus, its action should be undoable. This is an important point to remember when adding AppleScript support to an application. The user who runs a script that changes the document expects to be able to recover from any mistake by undoing it. AppleScript support must always honor the user's undo and redo expectations.

It turns out that the tasks of changing the object, making the change undoable, and setting a meaningful undo name can be accomplished using a block of reusable code. You therefore place it in a separate method taking three parameters: the range of text to be replaced, the text to substitute for the replaced text, and the undo action name.

Write this method first. Declare it at the end of the `DiaryDocument+VRAppleScriptSupport.h` header file like this:

```
#pragma mark APPLESCRIPT SUPPORT
```

```
- (void)updateDocumentInRange:(NSRange)range withText:(id)text  
    undoActionName:(NSString *)actionName;
```

Define it at the end of the implementation file like this:

```
#pragma mark APPLESCRIPT SUPPORT
```

```
- (void)updateDocumentInRange:(NSRange)range withText:(id)text  
undoActionName:(NSString *)actionName {  
    NSTextView *keyView = [[[self windowControllers] objectAtIndex:0]  
        keyDiaryView];  
    NSTextStorage *storage = [self diaryDocTextStorage];  
    if ([keyView shouldChangeTextInRange:range replacementString:text]) {  
        if ([text length] == 0) {  
            [storage deleteCharactersInRange:range];  
        } else {  
            if ([text isKindOfClass:[NSAttributedString class]]) {  
                [storage replaceCharactersInRange:range  
                    withAttributedString:text];  
            } else {  
                [storage replaceCharactersInRange:range withString:text];  
            }  
        }  
        [keyView didChangeText];  
        [[[keyView undoManager] setActionName:actionName];  
    }  
}
```


You must import the `DiaryWindowController.h` header file into the `Diary Document+VRAppleScriptSupport.m` implementation file. Add this to its imports list:

```
#import "DiaryWindowController.h"
```

You previously wrote code in Recipe 4 to change an `NSTextStorage` object with undo support, so none of this is new. Before changing the text storage object, you must test the waters by calling the text view's `-shouldChangeTextInRange:replacementString:` method. Then, if allowed to proceed, you must end the change by calling the view's `-didChangeText` method. Then you can set the undo action's name.

The change itself is handled between those two method calls. You're getting a little ahead of yourself here by including a clause that deletes text in the range. You will need this later, when you support AppleScript's `delete` command. There, you will pass in an empty text object of 0 length when you want to delete all the text in the specified range. Here, where you are replacing the text in the range, the incoming text object has a length greater than 0.

In addition to providing undo support, this method performs another important task in Vermont Recipes. It triggers the `-textStorageDidProcessEditing:` delegate method that causes the diary document to invalidate its `orderedEntries` array by reparsing the text storage.

You noticed earlier that the incoming text parameter to some of the setter accessor methods is untyped. This is because you cannot know at compile time whether a script will provide a rich text object or a plain text object to the set command, and you need to call different methods to deal with them. If it is a rich text object, you call `NSMutableAttributedString`'s `-replaceCharactersInRange:withAttributedString:` method. If it is a plain text object, you call `NSMutableAttributedString`'s `-replaceCharactersInRange:withString:` method.

When you run a script using one of the setter methods, this method actually changes the visible content of the Chef's Diary document before your eyes, marks the document dirty, and sets up the undo action. If you then click in the document and open the Edit menu, you see an appropriate command, such as `Undo Set Diary Document Text`, at the top.

7. Now write the first setter method, `-setName:`, immediately preceding the `-name` getter method in the `DiaryEntry.m` implementation file:

```
- (void)setName:(NSString *)name {
    SLOG(@"setname: %@", name);
    NSString *nameString = [@" " stringByAppendingString:name];
```

(code continues on next page)

```

NSRange nameRange = [[self diaryDocument] rangeOfLineFromMarkerIndex:
    [self entryRange].location];
nameRange.location += 1;
nameRange.length -= 1;
NSRange nameRangeInDocument = NSMakeRange([self entryRange].location
    + nameRange.location, nameRange.length);

NSString *actionName = NSLocalizedString(@"Set Diary Entry Name",
    @"name of undo action for AppleScript set name");
[[self diaryDocument] updateDocumentInRange:nameRangeInDocument
    withText:nameString undoActionName:actionName];
}

```

The incoming text to be substituted into the document in place of the replacement text is in the name parameter of the `-setName:` method. You add a leading space because it is inserted following the entry marker that signals the beginning of this entry in the diary document's text storage.

It sets the replacement range by adjusting for the additional space character.

It sets the undo action name to `@"Set Diary Entry Name"` using `NSLocalizedString()`.

Then it calls the new reusable method, `-updateDocumentInRange:withText:undoActionName:`, with these three values as its arguments to change the diary document's text storage, set up undo support, and trigger the update of the document's `orderedEntries` array.

8. The remaining setters are similar, and I leave them to you. They are fully implemented in the downloadable project file for Recipe 12. All of them call the `-updateDocumentInRange:withText:undoActionName:` method that you just wrote. Add `-setDate:` before `-date`, `-setTags:` before `-tags`, `-setBodyText:` before `-bodyText`, and `-setEntryText:` before `-entryText`. Also write the setter for the diary document's `-setDocumentText:` method, which you deferred when you wrote the getter in Recipe 4. You'll find it implemented in the downloadable project file for Recipe 12, too.

The `-setBodyText:` and `-setEntryText:` methods merit some discussion of an interesting feature of AppleScript's text handling. In the past, AppleScript was able to read and write styled text. Currently, AppleScript's styled text capabilities have largely been lost due to a lengthy process over several major releases of Mac OS X, wherein Apple converted AppleScript to full support of Unicode and unified all text handling into a single `text` data type.

The way you implement the setter methods here allows you to work with styles in a very limited fashion. If you extract rich text from a diary entry, you can use

it to set the text of another diary entry in the same document while preserving any styling, such as font, size, italics, and the like. But you can do that only within a single Apple event or by using AppleScript references. If you send two Apple events, such as when passing the text to another application, AppleScript converts the rich text to plain text and all styling is lost.

You can see this by comparing the results of these two scripts. Build and run the application first, and create two diary entries with some text. Add styling to the first entry by changing the font or its size, underlining it, or making similar changes. The first script is this:

```
tell application "Vermont Recipes"
    tell document 1
        set richTextRef to a reference to entry text of diary entry 1
        set entry text of diary entry 2 to richTextRef
    end tell
end tell
```

When you run it, diary entry 2 acquires the same text, with the same styling. This is the second script:

```
tell application "Vermont Recipes"
    tell document 1
        set richTextObject to entry text of diary entry 1
        set entry text of diary entry 2 to richTextObject
    end tell
end tell
```

When you run this version, diary entry 2 acquires the same text, but the styling is lost.

Step 7: Add a Current Diary Entry Property to the Document Class

In Step 3, you added the `current diary document` property to the `application` class to give scripts easy access to the diary document so that they can get at its properties and elements. There is only one diary document, and this is a convenient way to assist a scripter in getting it because it does not require knowing the diary document's name or its index.

The same idea can be applied to the diary entries in the diary document. The diary entry that currently contains the insertion point can be thought of as the `current`

diary entry. Creating a current diary entry property does not require any techniques you have not already seen, so I won't detail it here. It is implemented in the downloadable project file for Recipe 12.

In summary, you add a read-only **current diary entry property** element to the **diary document** class in the **sdef** file. You declare and define the **-currentDiaryEntry** accessor method in the **DiaryDocument+VRAppleScriptAdditions** class, adding an **@class** forward reference to the header file and importing the **DiaryEntry.h** and **DiaryWindowController.h** headers to the implementation file.

Step 8: Support the Make Command for New Diary Entries

In the remainder of this recipe, you will work with AppleScript commands, which you implement using a number of different Cocoa Scripting techniques.

In Steps 5 and 6, you arranged for a script to get and set the properties of an existing diary entry that the user had presumably typed into the diary document manually. In this step, you add the ability to create a new diary entry using AppleScript. Using the **make** command in the Standard Suite requires creating a new diary entry object with specified values and inserting it into the diary document's **diary entries** element.

In most cases, implementing the **make** command for an object in an AppleScript element like the Vermont Recipes **diary entries** element requires you to write only one new method, a KVC-compliant method described shortly that inserts the new object into the array underlying the AppleScript element.

It is important to know how this works behind the scenes, because you're going to modify the default behavior in this step. When a script runs the **make** command, Cocoa Scripting normally calls the default Cocoa implementation of the **-performDefaultImplementation** method in a built-in Cocoa class, **NSCreateCommand**. The Standard Suite's **sdef** file specifies that class as the Cocoa class for the **make** command—go ahead, look it up in the **CocoaStandard.sdef** file to see for yourself. That method then creates and initializes the new object by calling its **-init** method, and the method then calls your KVC-compliant method to insert the newly created object into the array. You can read about the **-performDefaultImplementation** method in the class reference for **NSScriptCommand**, which is the superclass of **NSCreateCommand**.

The KVC-compliant method you write here to make all this work is **-insertObject:in<Key>AtIndex:**, based on the **NSKeyValueCoding** informal protocol. As discussed below, it could instead be either **-insertIn<Key>:atIndex:** or **-insertIn<Key>:**,

based on the NSScriptKeyValueCoding informal protocol, depending on whether you allow the scripter to specify the location of the insertion or instead supply the location yourself.

This is not the first time you have seen the use of angle brackets to describe KVC methods. As you learned earlier, the `<Key>` portion is a placeholder for the name of your accessor method for the element, in this case `-orderedEntries`. Capitalization is governed by the capitalization used in the placeholder, either `<Key>` or `<key>`. Thus, in this case, the KVC-compliant method you write is `-insertObject:inOrderedEntriesAtIndex:` or, if you use the NSScriptKeyValueCoding informal protocol, either `-insertInOrderedEntries:atIndex:` or `-insertInOrderedEntries:`. You can read about these methods in the reference documents for the NSKeyValueCoding and NSScriptKeyValueCoding informal protocols, the *Cocoa Scripting Guide*, and the *Key-Value Coding Programming Guide*, which explain that Cocoa Scripting automatically calls one of these KVC-compliant methods if you implement it. Usage of the NSScriptKeyValueCoding informal protocol is also illustrated in Apple's *SimpleScriptingObjects* sample code project, and in fairly extensive comments in the NSScriptKeyValueCoding.h header file.

In Vermont Recipes, this mechanism won't work as is because the DiaryEntry class cannot be validly initialized using `-init`. NSCreateCommand's `-performDefaultImplementation` method calls `-init` on the newly created object, and as a result it won't properly set up the new DiaryEntry object. The DiaryEntry object requires a more complex initializer that takes arguments for the range of the diary entry in the diary document and a reference to the diary document.

To overcome this difficulty, in Leopard Apple exposed NSObject's `-newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` method. This is the method that NSCreateCommand's `-performDefaultImplementation` method calls to create a new object in response to the `make` command. By default, the `-newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` method calls the new object's `-init` method, but you may override it to call a more complex initializer of your object if `-init` is not a valid initializer. Apple exposed this method primarily because it is useful when adding AppleScript support to a Core Data application, where `-init` is not a valid initializer of the NSManagedObject class, but it is also useful for any object, like DiaryEntry, that requires a complex initializer. You can see the hoops through which a developer had to jump to make a Core Data application scriptable before Leopard by downloading my example code for the Wareroom Demo application, written for Mac OS X 10.4 Tiger, at http://www.quecheesoftware.com/Quechee_Software/Wareroom_Demo.html. The `-newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` method also makes it very easy to parse any `with data` or `with properties` parameter to the `make` command.

1. To implement the `make` command for the diary entry class, start by overriding the `-newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` method. Add it to the `DiaryDocument+VRAppleScriptAdditions.m` implementation file after the `-updateOrderedEntries` method, like this:

```
- (id)newScriptingObjectOfClass:(Class)class
    forValueForKey:(NSString *)key withContentsValue:(id)contentsValue
    properties:(NSDictionary *)properties {
    if (class == [DiaryEntry class]) {
        id fullText;

        NSDate *date = [properties objectForKey:@"date"];
        NSString *titleInsert = [self entryTitleInsertForDate:date];

        id entryText = [properties objectForKey:@"entryText"];
        if (entryText) {
            if ([entryText isKindOfClass:[NSAttributedString class]]) {
                fullText = [[[NSMutableAttributedString alloc]
                    initWithString:titleInsert] autorelease];
                [fullText appendAttributedString:entryText];
            } else {
                fullText = [NSString stringWithFormat:@"% %@%",
                    titleInsert, entryText];
            }
        } else {

            NSArray *tagsArray = [properties objectForKey:@"tags"];
            NSString *tags;
            if (tagsArray) {
                tags = [[self tagTitleInsert] stringByAppendingString:
                    [tagsArray componentsJoinedByString:@" "]];
                tags = [tags substringFromIndex:1];
                tags = [tags stringByAppendingString:@"\n"];
            } else {
                tags = @"";
            }

            id bodyText = [properties objectForKey:@"bodyText"];
            if (!bodyText) bodyText = @"";
        }
    }
}
```



```

        if ([bodyText isKindOfClass:[NSAttributedString class]]) {
            fullText = [[NSMutableAttributedString alloc]
                initWithString:titleInsert];
            if (tags) [fullText appendAttributedString:
                [[[NSAttributedString alloc] initWithString:tags]
                autorelease]];
            if (bodyText) [fullText appendAttributedString:bodyText];
        } else {
            fullText = [NSString stringWithFormat:@"%@@%@@%",
                titleInsert, tags, bodyText];
        }
    }

    tempDiaryEntryText = fullText;

    NSRange entryRange = NSMakeRange([[self diaryDocTextStorage]
        length], [fullText length]);
    DiaryEntry *entry = [[DiaryEntry alloc] initWithRange:entryRange
        forDocument:self];

    SLOG(@"newScriptingObjectOfClass: %@ forValueForKey:
        %@ withContentsValue: %@ properties: %@",
        NSStringFromClass(class), key, contentsValue, properties);
    return entry;
}
return [super newScriptingObjectOfClass:class forValueForKey:key
    withContentsValue:contentsValue properties:properties];
}

```

This isn't as daunting as it looks. It has a lot of work to do, but it proceeds in an orderly fashion.

First, in the outer `if` block, it checks whether the class to create is the `DiaryEntry` class. If not, it calls the superclass's implementation and exits.

Then it gets the `date` property, because the name of a diary entry is supposed to be its date. You will see when you get around to overriding `NSCreateCommand`'s `-performDefaultImplementation` method that it guarantees the availability of a date here by adding the current date to the `make` command's properties if the script did not include a date in a `with properties` clause.

Next, it gets the `entryText` property, if the script's `make` command provided one in a `with properties` clause, and combines it with the date. In *Vermont Recipes*,

the `make` command prefers an `entryText` property, and if one is found, it ignores any inconsistent `tags` and `bodyText` properties.

If the `make` command did not provide an `entryText` property, the method next gets any `tags` and `bodyText` properties and combines them with the date.

Finally, the method creates a new `DiaryEntry` object, initializes it with the designated initializer, `-initWithRange:forDocument:`, and returns it.

Notice that the returned object is not autoreleased. The basic Cocoa memory management rule is that methods beginning with certain key terms, including `new`, always return a retained object. The documentation for this method does not remind you of it, because you are supposed to remember it.

Near the end, after constructing the full text of the new diary entry, the method stores it temporarily in an external static variable, `tempDiaryEntryText`, so that it will be available a moment later to the `-insertObject:inOrderedEntriesAtIndex:` method. You could have modified the designated initializer to accept the entry text and place it in an instance variable for later use, but it is not uncommon in Cocoa to use a static variable when, as here, the value is temporary and needed nowhere else. Be sure to declare and initialize the static variable at the top of the file, before the `@implementation` directive, like this:

```
static id tempDiaryEntryText = nil;
```

2. Next, implement the `NSKeyValueCoding` informal protocol method.

I chose earlier to implement `-insertObject:in<Key>AtIndex:` instead of the `-insertIn<Key>:atIndex:` or `-insertIn<Key>: NSScriptKeyValueCoding` informal protocol method. The `NSKeyValueCoding` approach and the `NSScriptKeyValueCoding` approach are alternate techniques in much the same way that the two techniques for getting diary entries out of the `diary` `entries` element discussed in Step 5 were alternative techniques. Here, I rely on the `NSKeyValueCoding` informal protocol method because it is newer and because the “Maintain KVC Compliance” subsection of the *Cocoa Scripting Guide* tells you to implement the `-insertObject:in<Key>AtIndex:` and related `NSKeyValueCoding` methods that are described more fully in the *Key-Value Coding Programming Guide*. Another reason to prefer it is that the documentation makes clear that it invokes automatic key-value observing (KVO), which is essential if your application uses Cocoa bindings. Although the alternative `NSScriptKeyValueCoding` informal protocol methods apply specifically to AppleScript and they are used and described in Apple’s *SimpleScriptingObjects* sample code, they are older technology and their documentation does not currently indicate whether they invoke automatic KVO.

The `-insertObject:inOrderedEntriesAtIndex:` method takes an index. This would typically be derived from the `make` command's `at` parameter, which allows a script to specify a location for the insertion. Here, however, a new diary entry is always inserted at the end because diary entries are supposed to be kept in chronological order. Your implementation therefore ignores any index argument.

Add the method at the end of the AppleScript Support section of the `Diary-Document+VRAppleScriptAdditions.m` implementation file:

```
#pragma mark KVC METHODS SUPPORTING APPLESCRIPT ELEMENTS
#pragma mark orderedEntries

- (void)insertObject:(id)entry inOrderedEntriesAtIndex:(NSUInteger)index {
    SLOG(@"insertObject: %@, inOrderedEntriesAtIndex: %lu --
        IGNORING INDEX", entry, (unsigned long)index);
    [[self orderedEntries] insertObject:entry
        atIndex:[self orderedEntries] count]];

    NSRange entryRange = NSMakeRange([self diaryDocTextStorage] length,
        0);
    NSString *actionName = NSLocalizedString(@"Insert Entry",
        @"name of undo action for AppleScript make new diary entry");
    [self updateDocumentInRange:entryRange withText:tempDiaryEntryText
        undoActionName:actionName];

    tempDiaryEntryText = nil;
}
```

The method inserts the newly created `DiaryEntry` object at the end of the `orderedEntries` array using standard `NSMutableArray` techniques. It then uses the same technique you used when setting properties to update the diary document's text storage object with full undo support and force an update of the `orderedEntries` array to adjust the ranges stored in all the diary entries.

At the end of the method, you reset the `tempDiaryEntryText` static variable to `nil`. This isn't really necessary, but it is prudent because you might make some unanticipated use of the variable in a future version of the application.

The downloadable project file for Recipe 12 includes for the sake of illustration commented-out implementations of both of the `NSScriptKeyValueCoding` methods, the `-insertInOrderedEntries:` method to handle the case in which the script provides no `at` parameter, and the `-insertInOrderedEntries:atIndex:` method to handle the case in which the script provides an `at` parameter that Vermont Recipes should ignore.

3. Now you must subclass `NSCreateCommand` to make it deal correctly with the case where a script's `make` command does not provide a date in its `with properties` clause. You don't normally have to override `NSScriptCommand` or any of its subclasses, but there is no reason not to do so when necessary.

First, create a new pair of files in the AppleScript Support group of the Groups & Files pane, naming them **CreateCommand.h** and **CreateCommand.m**. You've done this enough times that detailed instructions should no longer be necessary. Be sure to change the superclass from `NSObject` to `NSCreateCommand`.

Override the `-performDefaultImplementation` method by adding this to the `CreateCommand.m` implementation file:

```
- (id)performDefaultImplementation {
    FourCharCode classCode =
        [[self createClassDescription] appleEventCode];
    if (classCode == 'dEnt') {

        NSAppleEventDescriptor *containerDescriptor =
            [[self appleEvent] attributeDescriptorForKeyword:'subj'];
        NSScriptObjectSpecifier *containerSpecifier =
            [NSScriptObjectSpecifier objectSpecifierWithDescriptor:
                containerDescriptor];

        if ([[containerSpecifier keyClassDescription] appleEventCode]
            == 'dDoc') {
            DiaryDocument *diaryDocument =
                [containerSpecifier objectsByEvaluatingSpecifier];
            if (diaryDocument != nil) {

                NSDictionary *resolvedKeyDictionary =
                    [self resolvedKeyDictionary];
                if ([resolvedKeyDictionary objectForKey:@"date"] == nil) {
                    NSMutableDictionary *tempDictionary =
                        [NSMutableDictionary dictionaryWithObject:
                            [NSDate date] forKey:@"date"];
                    [tempDictionary addEntriesFromDictionary:
                        resolvedKeyDictionary];
                    resolvedKeyDictionary =
                        [[tempDictionary copy] autorelease];
                }

                DiaryEntry *diaryEntry =
                    [diaryDocument newScriptingObjectOfClass:
                        [DiaryEntry class] forValueForKey:@"diary entry"
                        withContentsValue:nil
                        properties:resolvedKeyDictionary];
            }
        }
    }
}
```

```

        [diaryDocument insertObject:diaryEntry
         inOrderedEntriesAtIndex:0];
        NSScriptObjectSpecifier *containerRef =
            [diaryDocument objectSpecifier];
        NSScriptObjectSpecifier *specifier =
            [[NSNameSpecifier alloc]
             initWithContainerClassDescription:
             [containerRef keyClassDescription]
             containerSpecifier:containerRef
             key:@"orderedEntries" name:[diaryDocument
             entryTitleForDate:[resolvedKeyDictionary
             objectForKey:@"date"]]];
        SLOG(@"CreateCommand performDefaultImplementation,
             returns specifier: %@", specifier);
        return specifier;
    }
}
}
return [super performDefaultImplementation];
}

```

The method begins by getting the new object's class code and, if it isn't the code defined in the `sdef` file for the `diary entry` class, calls the superclass's implementation.

It then gets the container specifier for the new `DiaryEntry` object. Before Leopard, this could only be done in Cocoa by using a private Cocoa method, which is generally frowned upon because Apple can remove or change private methods at will. I was instrumental, along with others, in persuading Apple to expose the formerly private `-objectSpecifierWithDescriptor:` and related methods in Leopard, primarily to help in making Core Data applications scriptable.

If the container's code is `dDoc`, specified in the `sdef` file as the diary document class's code, the method goes on to get any date it finds in the `make` command's `with properties` clause, providing the current date if the script does not provide a date.

Finally, the method calls your `-newScriptingObjectOfClass:forValueForKey:withContentsValue:properties:` method and your `-insertObject:inOrderedEntriesAtIndex:` method, creates a name specifier to inform Cocoa Scripting of its containment hierarchy, and returns the specifier.

Because this method calls `-insertObject:inOrderedEntriesAtIndex:`, you must declare that method, which you didn't do previously. Add it at the end of the `DiaryDocument+VRAppleScriptSupport.h` header file:

```
- (void)insertObject:(id)entry inOrderedEntriesAtIndex:(NSUInteger)index;
```

You must also import several header files into the CreateCommand.m implementation file:

```
#import "AppleScriptLog.h"
#import "DiaryDocument.h"
#import "DiaryDocument+VRAppleScriptAdditions.h"
#import "DiaryEntry.h"
```

4. Finally, you have to inform Cocoa Scripting that, instead of using NSCreateCommand for the AppleScript `make` command, it should use your new CreateCommand class. The CreateCommand class will take care of using NSCreateCommand if the CreateCommand can't handle the `make` command.

The way I have done this is to abandon the XInclude mechanism for installing the Standard Suite in the Vermont Recipes sdef file. Instead, I copy it in its entirety from the CocoaStandard.sdef file and paste it into the VermontRecipes.sdef file. Then I change the `cocoa` element of the `make command` element so that it specifies CreateCommand as its class, rather than NSCreateCommand. Pasting the entire Standard Suite into your application's sdef file is a recognized technique when you need to modify the Standard Suite.

5. Your implementation of the AppleScript `make` command for a diary entry departs in several ways from its normal usage. There is nothing wrong with this. It is in the nature of AppleScript as an extensible language that terminology behaves differently based on the peculiarities of a specific application. However, you should document the differences by adding this `documentation` element to the sdef file, near the beginning of the `diary entry class` element after the `synonym` element:

```
<documentation>
  <html><![CDATA[
    <p>When making a new diary entry, it is always inserted at the
      end of the diary entries element, ignoring any
      <code>at</code> parameter. The <code>name</code>
      and <code>date</code> properties are automatically
      set to the current date if the script does not provide
      a different date. Use <code>with properties</code> to
      set a different <code>date</code> and to add
      <code>tags</code> and <code>body text</code> properties.
      Any <code>name</code> property is ignored. If an
      <code>entry text</code> property is provided, any
      <code>tags</code> and <code>body text</code> properties
      are ignored. A <code>with data</code> clause is always
      ignored.</p>
    <p class="label">examples</p>
```



```
<pre><code>make new diary entry with properties -
    {date:date "Monday, January 18, 2010 5:03:25 PM", -
    tags:{"dessert", "Snack"} body text:"Sweet!"}
make new diary entry with properties -
    {entry text:"Made some great cookies!"</code></pre>
]]></html>
</documentation>
```

6. Before moving along, consider how you would insert a new diary entry object into the `orderedEntries` element if you had not used an array. In Step 5, you learned that you can retrieve an object from an AppleScript element without using an array and an accessor method by implementing two indexed accessor methods, `-countOf<Key>` and `-objectIn<Key>AtIndex:`.

A similar choice is available when inserting an object into an AppleScript element. The `NSKeyValueCoding` protocol method `-insertObject:in<Key>AtIndex:` allows you to use an array or any other technique for managing the application's data.

I won't reproduce here the specific code that Vermont Recipes might use to dispense with an array of diary entries. However, the downloadable project files for the book include a version of the finished application that does not use an array. It is named *Vermont Recipes 2.0.0 - Recipe 14 Alternate*. To find all of the code changes in it that relate to this alternative way to support an AppleScript element, search the project for the phrase *RECIPE 14 ALTERNATE*. The project also includes the alternate technique for getting diary entries without using an array that I discussed in Step 5, and the technique for deleting diary entries that I will discuss in Step 9, next.

Step 9: Support the Delete Command for Diary Entries

The counterpart of the `make` command, which creates new objects and adds them to the application, is the `delete` command, which removes them from the application.

I won't spell out how to implement the `delete` command for diary entry objects here, but it is fully implemented in the downloadable project file for Recipe 12. All you have to do is implement another `NSKeyValueCoding` method, `-removeObjectFromOrderedEntriesAtIndex:`. It follows the `-removeObjectFrom<Key>AtIndex:` pattern described in the *Cocoa Scripting Guide* and the *Key-Value Coding Programming Guide*. You'll find it in the `DiaryDocument+VRAppleScriptAdditions.m` implementation file.

You could instead use the `NSScriptKeyValueCoding` protocol method `-removeFromOrderedEntriesAtIndex:`, based on the `-removeFrom<Key>AtIndex:` pattern. I won't reproduce here the specific code that this method might use. A commented-out implementation is included in the downloadable project file for Recipe 12 for illustration.

The downloadable project files for the book also include a version of the finished application in which the `-removeObject:from<Key>AtIndex:` method does not use an array. It is named *Vermont Recipes 2.0.0 - Recipe 14 Alternate*. To find all of the code changes in it that relate to this alternative way to support AppleScript, search the project for the phrase *RECIPE 14 ALTERNATE*.

You have already fully implemented the `-updateDocumentInRange:withText:undoActionName:` method to support deletion by testing its `withText:` parameter value to see whether it has a length of 0. Be sure to pass `@""` in that parameter in the `-removeObjectFromOrderedEntriesAtIndex:` method.

Step 10: Add a Custom Verb-First Command—Sort

The `make` and `delete` commands in the Standard Suite are known as *verb-first* commands. You create a command object first, such as `CreateCommand`, and only then do you worry about communicating it to the object on which it acts. There is another kind of command known as an *object-first* command. You will implement an object-first command in Step 11.

The *Cocoa Scripting Guidelines* explain when to use one or the other of these techniques.

Object-first commands are best for relatively simple actions that don't require much information about other objects, such as reversing the state of a single object, and that don't involve much processing overhead. You implement a method in an existing object—that's why it's called an object-first command—that performs the command. Typically, such methods might be named something like `-handleEncryptCommand:` and its reverse, `-handleDecryptCommand:`. The handler methods for object-first commands may be called repeatedly, once for each object to which they are addressed.

Verb-first commands are more appropriate for interaction between multiple objects, especially if significant processing overhead is involved. The standalone command object is called once, and it performs all of the processing internally.

In this step, you implement a completely new verb-first command, unique to *Vermont Recipes*. It is the `sort` command, which a script can execute to sort all of the diary entries in the diary document into chronological order. It can sort the diary entries in ascending or descending order, just so that you see how a parameter is added to

a command, although sorting the diary entries in descending order doesn't make much sense in light of the Vermont Recipes specification.

You have modified two verb-first commands that are implemented in Cocoa Scripting already, so this step won't be particularly new to you. Basically, you write a subclass of `NSScriptCommand` and override its `-performDefaultImplementation` method.

1. Start, as usual, by editing the `sdef` file. For the `sort` command, you need to add its `command` element. You also need to add an `enumeration` element to define the type of its optional `sort order` parameter, which contains one of two values, `ascending` and `descending`. The AppleScript direct parameter is the diary document.

Unlike most object-oriented languages, AppleScript's commands are global in the sense that you implement them at the top level of the terminology suite, not inside a `class` element. A `responds-to` element can be added to individual `class` elements to indicate that a class responds to a particular command.

Add the following elements near the top of the Vermont Recipes Suite in the `sdef` file, after the `saveable file format` enumeration:

```
<enumeration name="sort order" code="VRso">
  <enumerator name="ascending" code="VRag"
    description="Ascending sort order.">
    <cocoa integer-value="0"/>
  </enumerator>
  <enumerator name="descending" code="VRdg"
    description="Descending sort order.">
    <cocoa integer-value="1"/>
  </enumerator>
</enumeration>

<command name="sort" code="VRdydDst" description="Sort a document.">
  <cocoa class="SortCommand"/>
  <direct-parameter description="The document to sort.">
    <type type="document"/>
  </direct-parameter>
  <parameter name="in order" code="dEso" type="sort order"
    description="The sort order (ascending by default).">
    optional="yes">
    <cocoa key="sortOrder"/>
  </parameter>
</command>
```

The `sort order` enumeration uses a new technique, introduced in Tiger, which allows you to specify a type and a value for individual `enumerator` elements.

Previously, you had to give each enumerator a four-character code and echo that code in a C `enum` declaration in your code. Now you can specify an integer, a Boolean value, or something else and supply the value for each in the `sdef` file. Here, you use an integer type, and your code will do the same with a more traditional integer enumerator value.

The `cocoa` subelement of the `command` element specifies that you will create a `SortCommand` class in code. The `direct-parameter` element specifies that it takes a document. The `parameter` element—there can be more than one in a `command` element—specifies that it will be identified in a script by the term `in order`, that its type is `sort order`, and that it is optional. The `sort order` type is the enumeration having that name. In combination, these elements indicate that a `sort` command takes a direct parameter that is a document and an optional parameter using the term `in order`, followed optionally by one of the two enumerators, `ascending` or `descending`. When you implement an optional parameter like this, you should always describe the default value for a scripter's benefit, as you do here in the parameter's `description` attribute. This command does not return a result.

You specify a document as the `sort` command's direct parameter in anticipation of applying the command not only to the diary document and its diary entries, but to the recipes document and its contents. By generalizing the command and allowing it to take any kind of document as a direct parameter, you maximize the flexibility of the application's custom terminology while minimizing the number of terms required. When you code the `sort` command, you will test to make sure the receiver is in fact a diary document. Eventually, you anticipate adding a test for a recipes document and writing code appropriate to that type of document.

There is one other `sdef` requirement for the `sort` command: You must add a `responds-to` element in the class that responds to it. Add this at the end of the `diary entry` element:

```
<responds-to command="sort">  
  <cocoa method=""/>  
</responds-to>
```

In the case of a verb-first command like `sort`, the method you specify is an empty string. You don't specify `"performDefaultImplementation"` because that is assumed in all commands based on `NSScriptCommand`. If you omit this `responds-to` element, a script using the `sort` command won't work unless you place the direct parameter in parentheses and precede it with a `get` command. When you include this `responds-to` element, a script can be written directly, such as `sort diary entries of current diary document`. This technique is not currently explained in the Cocoa Scripting documentation.

Eventually, you anticipate adding an identical **responds-to** element to a new **recipes document** element so that it can be sorted by the same command.

2. Now create the `SortCommand` class. This is similar in concept to the `CreateCommand` class you wrote in Step 8, so I won't reproduce it in full here. It is fully implemented in the downloadable project file for Recipe 12.

In summary, create new `SortCommand.h` and `SortCommand.m` source files, and place them in the AppleScript Support group in the Groups & Files pane in the Xcode project window. Change the `@interface` directive to this:

```
@interface SortCommand : NSScriptCommand {
```

Override the `-performDefaultImplementation` method as shown in the `SortCommand.m` implementation file in the downloadable source file for Recipe 12. It parses the command's parameters, and based on their values, it performs the sorting operation in ascending or descending order. It does this by comparing the titles of two date entries in the diary document's text storage, which should represent valid dates, and sorting them using one of several array sorting methods provided by `NSMutableArray`. Snow Leopard has added some blocks-based sorting methods, one of which is used here if Snow Leopard is running. The method then uses a standard technique to build a new temporary text storage object in the proper order and to update the **orderedEntries** array with the new ranges. Undo is fully supported. The method returns `nil` because the command does not specify a result.

The code that performs the sort operation is not the point of this recipe. Instead, focus on how the method obtains and evaluates the direct parameter and the optional parameter of the command. `NSScriptCommand` provides several methods to obtain and evaluate command parameters, and you should experiment to become familiar with all of them.

Here, the `-directParameter` method returns an object specifier for the document. You then use the `NSScriptObjectSpecifiers` `-objectsByEvaluatingSpecifier` method to get the actual document object. You test its class to determine whether it is `DiaryDocument`, and if so, you call its `-orderedEntries` method to get the array of diary entries. The script sorts the entire **orderedEntries** array.

You also use the `NSScriptObjectSpecifiers` `-arguments` method to get the arguments from the script. In this case, the `in order` parameter is optional, and it was declared in the `sdef` file to use the key `sortOrder`. Using this key, you get the value of the `sortOrder` parameter from the `NSDictionary` object returned by the `-arguments` method. If the script did not provide a value in the optional `in order` parameter, there will be no `sortOrder` entry in the dictionary, the attempt to get it will therefore return `nil`, and the code interprets this as a `sortOrder` parameter of 0. This is the default `ascendingOrder` value specified in the `sdef` file. If the script did provide an `in order` parameter value, either `ascending` or `descending`, then you get it in the form of an `NSNumber` object, from which you extract its `-intValue`.

Note that the `sortOrder` enum is declared in the `SortCommand.h` header file.

The final piece of this puzzle is the comparison methods used by the Leopard version of the sort operation to compare any two diary entries in aid of the sort operation. The Snow Leopard version doesn't need a separate comparison method, because it performs the comparison in a block defined within the statement itself. For Leopard, however, you need separate methods, `-compareDiaryEntry:` and `-reverseCompareDiaryEntry:`. They appear in the `DiaryEntry` class in the downloadable source file for Recipe 12. They call `NSDate`'s `-timeIntervalSinceDate:` method because `Vermont Recipes` doesn't need the millisecond accuracy of `NSDate`'s `-compare:` method.

Note that the `SortCommand.m` implementation file requires some import statements.

Step 11: Add Custom Object-First Commands—Encrypt and Decrypt

The final commands you will implement in this recipe are two object-first commands, `encrypt` and `decrypt`. These operations are appropriate for an object-first approach because they apply to a single diary entry and they reverse its state with relatively little overhead.

I embarrass myself here by using the ROT13 algorithm, a simple Caesar or substitution cipher that offers no meaningful security. Since the ROT13 algorithm is reversible, you actually only need a single command, `encrypt`, because applying it a second time decrypts the text. I implement both methods because you would need both if you used a nonreversible algorithm—for example, a Caesar cipher that shifts the alphabet by some number other than 13.

Because this is an object-first command, you don't have to create a subclass of `NSScriptCommand`. Instead, you add two new methods to the `DiaryEntry` class, `-handleEncryptCommand:` and `-handleDecryptCommand:`.

1. Add the `command` class elements to the `sdef` file following the new `sort` command element:

```
<command name="encrypt" code="VRdydEec"
    description="Encrypt the body text of a diary entry.">
  <cocoa method="handleEncryptCommand:"/>
  <direct-parameter description="The entry to encrypt.">
    <type type="diary entry"/>
  </direct-parameter>
</command>
```



```

<command name="decrypt" code="VRdydEdc"
    description="Decrypt the body text of a diary entry.">
    <cocoa method="handleDecryptCommand:"/>
    <direct-parameter description="The entry to decrypt.">
        <type type="diary entry"/>
    </direct-parameter>
</command>

```

Each takes a direct parameter consisting of a diary entry object. The `cocoa` element's `method` attribute specifies the command to be executed.

You also need two `responds-to` elements specifying the same methods. Add them to the end of the `diary entry class` element:

```

<responds-to command="encrypt">
    <cocoa method="handleEncryptCommand:"/>
</responds-to>

<responds-to command="decrypt">
    <cocoa method="handleDecryptCommand:"/>
</responds-to>

```

2. Write the encryption method, which for the ROT13 algorithm does double duty as the encryption and decryption method. This makes a good category on `NSMutableString`, adding the ability to encrypt all of your mutable strings, wherever desired. You created two categories in Recipe 7, `NSScreen+VRScreenAdditions` and `NSDrawer+VRDrawerAdditions`, and another two categories in this recipe, so I won't walk you through the process. Look at `NSMutableString+VRMutableStringAdditions` in the downloadable source file for Recipe 12 for details. I get enough of a kick out of it to reproduce the method's implementation here, but I'll let you figure out how it works. Declare it at the end of the `DiaryEntry.h` header file like this:

```

- (void)VR_rotateStringBy13:(NSMutableString *)string {
    static NSString *alphabet =
        @"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    static NSString *rotation =
        @"NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm";
    NSUInteger index;
    for (index = 0; index < [string length]; index++) {
        NSRange indexRange = NSMakeRange(index, 1);
        NSString *indexString = [string substringWithRange:indexRange];
        NSRange lookupRange = [alphabet rangeOfString:indexString];

```

(code continues on next page)

```

        if (lookupRange.location != NSNotFound) {
            NSString *lookupString =
                [rotation substringWithRange:lookupRange];
            [string replaceCharactersInRange:indexRange
                withString:lookupString];
        }
    }
}

```

Don't forget to import the new category's header file into `DiaryEntry.m`.

As written, the method makes no attempt to preserve rich text.

3. The `-handleEncryptCommand:` and `-handleDecryptCommand:` methods, along with a `-bodyTextRange:` utility method that they use, are declared and implemented in the AppleScript Support section of the `DiaryEntry` class in the downloadable project file for Recipe 12. They simply apply the `VR_rotateStringBy13` method to the body text of the specified diary entry, then display the encrypted or decrypted text, update the `orderedEntries` array (which isn't actually necessary in this case), and provide undo support.
4. One thing is missing from this step. There should be some way that a script can determine whether a text passage is currently encrypted before applying either encryption or decryption. Ideally, this would take the form of a Boolean property named `encrypted`. Its value must survive quitting and relaunching the application, so it should be saved with the document in some way. This could be done simply in Vermont Recipes by adding the unencrypted word *ENCRYPTED* to the beginning of an encrypted passage and removing it when the passage is decrypted, although this, like so much else in the diary document, would put it at risk of removal by manual editing. But I'm out of space, so I'll leave this as an exercise for the reader. It is *not* implemented in the downloadable project file.

Step 12: Move Along

You have now covered most of the ground required to add a decent level of support for AppleScript to an application. But there is much more to learn in order to take AppleScript support all the way. For example, I have omitted any discussion of AppleScript error reporting.

It isn't that AppleScript support is hard. With decent documentation, it is no harder than most tasks in Cocoa and easier than some. Instead, it's that there is so much to take care of. To do it right and be thorough about it, you should cover all of the

functionality of your application, and you should even consider adding some functionality via AppleScript that isn't available through the application's graphical user interface. AppleScript is a full-fledged user interface in its own right. You should take the time required to review every single operation that your application can perform for possible inclusion in the AppleScript interface, and more.

In addition, you have to test every operation that your AppleScript support provides, and you have to test all of them in all of the myriad ways in which AppleScript can do it. It isn't enough, for example, to satisfy yourself that `get document "Get Rich Quick"` works. You also have to test `get document 1` and `get every document` and `get first document whose first paragraph contains "script"`, and on and on. When I was testing my Wareroom Demo project, I wrote and ran several hundred test scripts.

The downloadable project file for Recipe 12 includes a folder named *Vermont Recipes Test Scripts*. This only scratches the surface of what you should do, but you may find it useful in understanding how the AppleScript support that you have added to Vermont Recipes works in practice. I have tried to provide a test script for every feature you created in this recipe. They are organized in subfolders based on the nature of the script.

There is also a subfolder with a few scripts that don't work. That folder is more useful than the rest, because it tells you where you should direct your attention to improve the application. As the AppleScript dictionary viewer tells you when you've done something wrong, "Nothing to see here; move along."

Step 13: Build and Run the Application

Build and run the application, then try out all of the test scripts in the Vermont Recipes Test Scripts folder in the downloadable project file for Recipe 12. Then combine them into more complex scripts that begin to look like real automated workflows. You can even write scripts that get text from one application, such as TextEdit, and place it in Vermont Recipes, and vice versa.

To do this kind of testing, you first have to add some data to the Chef's Diary document. For increased realism, use the Add Entry and Add Tag buttons in the diary window to create the skeletons of a dozen or more diary entries; then copy and paste very long text passages from some other document into all of the diary entries, and save the document for testing. Only then will the text scripts get a realistic workout.

Be sure you also study the Vermont Recipes terminology dictionary. Consider not only what you have created in this recipe, but also what you could add to make it even more useful. Think about its design, too, particularly whether its terminology and

organization could be more effective or understandable. This is a thought process that you should go through in the course of adding AppleScript support to any application.

Finally, try the scripts that don't work, and consider how you might make them and other scripts work properly. This will lead you back into the documentation and the many AppleScript-related classes in the Cocoa frameworks that you haven't touched upon in this recipe.

Step 14: Save and Archive the Project

Quit the running application. Close the Xcode project window, discard the build folder, compress the project folder, and save a copy of the resulting zip file in your archives under a name like **Vermont Recipes 2.0.0 - Recipe 12.zip**. The working Vermont Recipes project folder remains in place, ready for—what? Could it be? Are you done with the Vermont Recipes application?

Conclusion

Well, no, I find that an application is never finished. I always have ideas that I haven't yet gotten around to implementing, and the to-do list just keeps on growing. That's a good thing, because there's nothing like a new release of an application every few months to drive increased sales.

But you have to pull it together into a working product every once in a while, or you'll have no sales at all. In the case of Vermont Recipes, of course, it isn't really a working product. The Chef's Diary is in very nice shape, and it has served well as a platform to teach a great many of the steps that go into writing any complete application. But the recipes document is still represented by nothing more than a window with a half-empty toolbar and three completely empty panes. This application is definitely not yet ready for prime time.

In this case, however, the product is this book, and it is almost finished. In Recipe 13, I'll wrap up the process with a discussion of what you can or must do to release a finished application into the marketplace, and in Recipe 14 I'll get you started on converting the application to use more modern techniques such as properties, Cocoa Bindings, and garbage collection.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 12.

Class Reference and Protocol Documents

NSObject Class Reference (several methods for AppleScript support)

NSAppleEventDescriptor Class Reference

NSScriptingComparisonMethods Protocol Reference

NSScriptKeyValueCoding Protocol Reference

NSScriptObjectSpecifiers Protocol Reference

NSWindowScripting Protocol Reference

NSClassDescription Class Reference

NSScriptClassDescription Class Reference

NSScriptCoercionHandler Class Reference

NSScriptCommand Class Reference

NSScriptCommandDescription Class Reference

NSScriptExecutionContext Class Reference

NSScriptSuiteRegistry Class Reference

NSScriptCommand Class Reference

NSCloneCommand Class Reference

NSCloseCommand Class Reference

NSCountCommand Class Reference

NSCreateCommand Class Reference

NSDeleteCommand Class Reference

NSExistsCommand Class Reference

NSGetCommand Class Reference

NSMoveCommand Class Reference

NSQuitCommand Class Reference

NSSetCommand Class Reference

NSScriptObjectSpecifier Class Reference

NSIndexSpecifier Class Reference

NSMiddleSpecifier Class Reference

(continues on next page)

DOCUMENTATION *(continued)*

Class Reference and Protocol Documents *(continued)*

NSNameSpecifier Class Reference

NSPositionalSpecifier Class Reference

NSPropertySpecifier Class Reference

NSRandomSpecifier Class Reference

NSRangeSpecifier Class Reference

NSRelativeSpecifier Class Reference

NSUniqueIDSpecifier Class Reference

NSWhoseSpecifier Class Reference

NSScriptWhoseTest Class Reference

NSLogicalTest Class Reference

NSSpecifierTest Class Reference

General Documentation

AppleScript Overview

AppleScript Language Guide

Cocoa and AppleScript: From Top to Bottom

Cocoa Scripting Guide

Technical Note TN2106: Scripting Interface Guidelines (the SIG)

Mac OS X Developer Tools Manual Page for sdef(5)

Mac OS X Leopard Developer Release Notes: Cocoa Foundation Framework

AppleScript Release Notes

AppleScript Terminology and Apple Event Codes Reference

Key-Value Coding Programming Guide

NSScriptKeyValueCoding.h (header file comments)

Sample Code

SimpleDefinitions (including Text Suite in Sketch.sdef)

SimpleScripting

SimpleScriptingObjects

SimpleScriptingProperties

SimpleScriptingVerbs

SimpleScriptingPlugin

Deploy the Application

With this recipe, you complete Section 2. You haven't implemented any support for the recipes document that forms the heart of the Vermont Recipes application specification, but the Chef's Diary is complete. For purposes of this book, you now have a working application.

If you were writing the application just for the fun of it, you're done. Well, you presumably want to use it yourself, so you should build its release configuration and place the finished application in your Applications folder, as described in Step 1. Then you'll be ready to move on to your next project.

If you're like most developers, however, you have greater ambitions. In one way or another, you want the application to be used by other people. Maybe you'll just give it to a few friends or post it on a Web site for download as freeware. Or maybe you have a business model in mind, and you will try to make money from it by distributing it as shareware or as a commercial product. In any of those cases, you have some more work to do.

First, of course, you must build the application for release so that it can run without the aid of Apple's developer tools. After that, you should put some effort into testing, documentation, and distribution formats and media. In addition, especially if you plan to charge a price, you should consider protecting your intellectual property against piracy, undertaking a marketing effort, and providing after-sale support.

I will touch on all of these topics in this recipe, not exhaustively but with a broad brush to point you in a useful direction. There is no code in this recipe.

HIGHLIGHTS

Building an application for release

Testing an application

Writing documentation

Providing support

Packaging an application for distribution

Using registration and transaction processing services

Promoting an application

Step 1: Build the Application for Release

Before proceeding, be sure to increment the build version. Leave the archived Recipe 12 project folder in the zip file where it is, and open the working Vermont Recipes subfolder. Increment the Version in the Properties pane of the target's information window for both the Vermont Recipes and the Vermont Recipes SL target from 12 to **13** so that the application's version is displayed in the About window as *2.0.0 (13)*. You aren't actually making any changes in the application in this recipe except to build it for release, but I like to increment the build version for the final release just to mark it as a clean version. Last-minute testing may reveal the need for a minor tweak here or there.

While you're at it, open the Info.plist file for both targets and the InfoPlist.strings file, and change the copyright notice for the CFBundleGetInfoString (Get Info String) and NSHumanReadableCopyright (Copyright [human-readable]) keys to extend the copyright into 2010, since we're into the new year already.

Until now, you have generally built the application using the project's Debug configuration. Among other things, the Debug configuration is typically set up to include code that makes it possible to use the debugging tools that Apple provides to developers. The optimization level is typically set relatively low because it speeds up compilation and you aren't concerned about execution speed during development.

The application's final release version involves different considerations. You don't need the debugging code that the Debug configuration supplies, and you don't want it because it slows the application down. Also, the Debug configuration typically peppers the console with debugging messages, most of which aren't appropriate for public consumption. Finally, some of the debugging code in the Debug configuration might make it easier for others to figure out how you wrote the application. In addition, you generally want to set the optimization level of the released application high enough to provide the best possible speed consistent with considerations such as memory footprint.

When you build the Release configuration, don't assume there will be no warnings just because you have succeeded in removing all warnings during compilation of the Debug configuration. Some warnings are generated only when optimization is turned on, so examine the build results and fix any new problems that crop up.

Review the "Building for Release" section of the *Xcode Project Management Guide* and the other documentation cited there for details on Xcode build settings for the Release configuration. One important topic that is not covered there is code signing, which you should read about in Apple's *Code Signing Guide* and related documentation. Code

signing was introduced in Mac OS X 10.5 Leopard. Your application will work if you don't sign it—at least for now—but it will trigger annoying security-related alerts when your users launch it. Apple “highly” recommends that you sign all code intended for use with Mac OS X 10.5 or newer.

In the latest versions of Xcode, building the Release configuration is as easy as opening the Overview pop-up menu once or twice and choosing Build > Build. If you are comfortable with the default build settings for the Release configuration—and in most cases you should be—just choose Release instead of Debug as the Active Configuration and build the project. To be completely sure the built product is correct, you might want to clean the project immediately before building it to clear any lingering discrepancies in the intermediate build products. You don't have to change the Active Architecture setting, because the build settings for the Release configuration control the architectures that the application supports.

One more step is required for Vermont Recipes, because you have Leopard and Snow Leopard targets. In addition to choosing the Release configuration, choose Vermont Recipes SL as the Active Target and build the project. When you choose the Active Target, Xcode automatically chooses the corresponding Active Executable. Then remove the built application from the Release subfolder of the build folder or add (**Snow Leopard**) to its name to avoid overwriting it when you build the other target. Then choose Vermont Recipes as the Active Target and build the project again, and then add (**Leopard**) to its name. If you don't label them by operating system version, be sure to save the two versions of the built application in separate folders properly labeled to tell them apart.

Examine each built application using the Finder's Get Info command. You see that the Kind of the Snow Leopard build is *Application (Intel)*, because Snow Leopard does not run on PowerPC computers. The Kind of the Leopard build of the application is *Application (Universal)* because it can run under Snow Leopard on Intel computers and under Leopard on both PowerPC and Intel hardware.

Step 2: Test the Application

You've been testing the Vermont Recipes application at the end of every recipe and, in many cases, at the end of individual steps. But you ought to do more testing.

During development, you can use unit testing. I have not covered unit tests in this book. Read the *Xcode Unit Testing Guide* and *Test Driving Your Code with OUnit* for more information. Near the end of the development cycle, run the release build with the Console open so that you can catch non-fatal errors that might have gone undetected until now.

Once the application is completed, you should test it in a workaday setting with an eye on its overall usability and correctness. Testing during development by looking for problems with your implementation of a particular feature just isn't enough.

Commercial applications are available to do automated testing. I can't endorse any of them because I haven't used them. Squish from froglogic at <http://squish.froglogic.com> and eggPlant from TestPlant (formerly Redstone Software) at <http://www.testplant.com/> are established players in this market.

If your application supports AppleScript, you can do automated testing yourself by putting together a suite of test scripts to exercise all of the scriptable operations that your application performs. This is especially useful for performing repetitive operations thousands of times overnight looking for slow-developing bugs. Testing with AppleScript has the advantage of testing your AppleScript implementation as well, and it may suggest ways in which you should improve your AppleScript support.

Even if your application is not scriptable in its own right, you can use Apple's GUI Scripting technology to automate testing by controlling your application's user interface elements such as menus and buttons.

In addition, you should enlist other human beings besides yourself to test your application. Your roommates, children, parents, friends, and neighbors might be able to help. Any one tester, you included, undoubtedly has particular ways of using a computer. I make very heavy use of the mouse and avoid keyboard shortcuts even though I'm a very fast touch typist, whereas you may use keyboard shortcuts almost exclusively while hunting and pecking at the keys. I use the menu bar a lot, whereas you may use contextual menus most of the time. All of us have different habits, and any of us may overlook bugs that others will likely find.

For quite some time, it has been very common to put free beta versions of an application out on the Web for public testing. When the practice first developed, many felt it was presumptuous to expect others to do free beta testing. Now, however, the practice is well established, and many people download free beta versions of applications and report any bugs they find. You could even buy a beta version of this book at the Safari Books Online Rough Cuts site. If you choose to do public beta testing of your application, my only advice is to label the application clearly and prominently as a beta version. It's only fair to your users, and it might even save you from a lawsuit if the beta version of your application erases somebody's hard drive. If you do public beta testing of an upgrade to your product, make sure that the last stable release version remains available at least until the beta test is over and you have released the new version. If your beta test is so public that you allow it to appear on sites like VersionTracker and MacUpdate, be sure to take control of the description of your product that those sites publish, and in particular make it very clear that it is a beta version. Both sites provide a way for developers to log in for free to write or edit a product's description.

The sad truth is that every application has bugs even if it isn't labeled "beta." So take advantage of every bug report you receive to do additional testing and to prepare fixes for the next release. Bug-tracking systems are available, but you don't really need anything more elaborate than a good list. Treat all of your customer feedback as if it were a beta tester's report.

Step 3: Provide Documentation

Mac OS X applications are famous for the consistency of their user interfaces, which is strongly encouraged by Apple in many ways. Using the Cocoa frameworks, particularly the AppKit, as the basis of your application and designing and building its user interface with Interface Builder ensure that users will be comfortable using your application from the outset, because it will conform to established Apple user interface guidelines detailed in the *Apple Human Interface Guidelines*. But don't rely only on the tools, because they contain many gaps. Part of your job as a developer is to become familiar with the documentation and with undocumented conventions and practices, and to follow them unless you have a carefully considered reason to do something different.

Nevertheless, you should always provide documentation. Sure, many users will never read it, but many other users will complain if you don't provide it. They will hound you for support. Worse, they may turn to a competing product that does offer good documentation.

In this book, you have learned how to provide four forms of documentation inside your application package. By putting documentation there, you ensure that it automatically follows your application wherever users install it. In addition, users can always find it by using the Help menu, by pausing the mouse over a user interface element, and, in the case of AppleScript support, by using AppleScript Editor's dictionary viewer or the similar dictionary viewer in another script editor such as Script Debugger. In Recipe 5, you included a read-me file in the application's Help menu. In Recipe 8, you added help tags to user interface elements. In Recipe 11, you added a Help book accessible through the Help menu's Vermont Recipes Help menu item. And in Recipe 12, you implemented AppleScript support, in part by creating an `sdef` file that documented itself in the form of a human-readable terminology dictionary.

That is not enough for any but the simplest of applications. If your application knows only a trick or two, a well-written Help book that covers the ground can be sufficient, although a surprising number of people never think to look in the Help menu. For anything that is even remotely complicated, I recommend that you write a separate document. Save it as a PDF file, because PDF files are universally readable. Provide it to users on your application's Web site, so that they can download it without the application if they want to see what your application does. Include it

in the application package, of course, but consider putting a copy of it or an alias file pointing to it on your distribution disk image, so that they can read it (and the installation instructions you include in it) before they launch the application. If it contains more information than the Help book, include it in the application package and connect a menu item in the Help menu to it.

Pay attention to spelling, grammar, and style. You may not think them important, but I assure you that many of your customers do. Rightly or wrongly, poorly written documentation convinces many potential customers that your code is probably also carelessly written. Even if you are a good proofreader, use a spelling and grammar checking application. If you are impatient with proofreading, get somebody to help. Be sure to follow the rules laid out in the latest version of the *Apple Publications Style Guide* to ensure consistency with the terminology and style used in Apple's documentation. Finally, put your documentation through a beta testing cycle. Typically, a developer is too close to the code to appreciate the full extent of a new user's ignorance, and beta testing the documentation will likely catch omissions.

As you learned in Recipe 11, an application's Help book should be relatively short and simple, without a lot of detail. But there are likely to be users of your application who want more handholding or more detail. Your application may contain features that aren't obvious from a cursory examination of its user interface, so explain them to your users. This is not just helpful to your users—it is good advertising.

Step 4: Provide User Support

No matter how much effort you put into your application's documentation, there will inevitably be users who can't find it, won't read it, don't understand it, or find holes in it. If you would rather be a hermit, go ahead and hide from them—and suffer the consequences. Otherwise, make an effort to be responsive to your users. It's good advertising, and it tends to prevent bad advertising by disgruntled customers.

If, like me, you're a solo developer and success hasn't yet put you in a position to need or afford employees, you won't want to run a help desk. You don't have to make your telephone number public—I do, but I've only received two calls with software questions in several years despite having thousands of customers.

Instead, include your e-mail address or at least your Web site's address in your application's About window, in its documentation, and on your application's Web site if you maintain one. Also, monitor all mailing lists where your application is likely to be discussed. It is even a good idea to run a Google search periodically to pick up discussions of your product that you would not run across on the mailing lists or blogs you normally read. A particularly good technique is to set up Google Alerts for your application's name and common misspellings of it, so that you will more

promptly discover discussions on forums, blogs, Twitter, and the like. Read about Google Alerts at <http://www.google.com/support/alerts/>.

Above all, answer your e-mail and respond to mailing list inquiries promptly, courteously, helpfully, and thoroughly. If you can't provide a full answer immediately, reply immediately anyway and explain that it will be a few days before you are able to respond in detail. I make a habit of doing all of these things, and I constantly get thank-yous for it. Also, my applications are offered in free 30-day trial versions, and I can tell you that providing good responses to questions from trial users results in sales.

Maintaining a Web site for your software is not only a good way to advertise your application, but also a good way to provide support. Be sure to list the URL in your application's About window, in its documentation, and in the signature of every computer-related e-mail message you send, whether it is about your product or not. Web sites are easy to create and inexpensive to maintain. A Web site is of course a good sales tool and a good way to make your application available for download, but it will also reduce the number of support messages you get if you include additional documentation on it. In fact, make sure there is a link on your Web site to a page where you can update help information based on the support messages you receive, and consider making this the primary link to your product in the About window and help documentation.

It has become increasingly easy to include interactive features on a Web site. If you have the time to learn how to do it, don't stint. Set up a discussion group on your Web site and monitor it closely. Respond to any message that reflects a problem or misunderstanding. If you have lots of ideas and like to talk about them, run a blog related to your application and provide your readers with a means to comment. Monitor the comments closely and respond when appropriate.

I don't have a big interest in Twitter, Facebook, MySpace, LinkedIn, and the like, but I have no doubt that social networking services, too, offer ample opportunity to provide support and to promote your software.

Step 5: Distribute the Application

If you don't put your application out there, nobody will use it. But distribution isn't necessarily easy. It requires you to make a number of choices and maybe even to do some hard work.

First, you must decide whether to give it away for free, to offer it as shareware, or to sell it as commercial software.

Free is easy. Put it on your Web site for download and be done with it. You don't even have to do that. Instead, you can open a free developer account on VersionTracker or

MacUpdate and post it there. You don't need to take steps to fend off pirates, because it's free and you don't care who copies it. Obviously, you don't need to set up registration keys or arrange to handle credit cards and foreign exchange transactions.

But you do have to put the application into some format that is suitable for download and distribution. This can be as easy as compressing the application package. Put the application and other files into a folder on your desktop, select the folder in the Finder, and choose File > Compress "My Application." The Finder immediately places a new MyApplication.zip file adjacent to the folder, and you can distribute the zip file. Users uncompress it by double-clicking the zip file.

For a long time, Apple has recommended that you distribute software on disk images, and most applications come on disk images now. The user mounts a disk image on the desktop by double-clicking the file, or if you follow Apple's recommendation and make it an Internet-enabled disk image, it unpacks and mounts itself when you download it. Once mounted, it looks and acts as if it were a real disk. There are some advantages to disk image files. For one thing, the application that opens and mounts disk images can be set up so that it displays your license agreement and requires the user to accept it before proceeding to mount the image. Also, you can put a pretty background image on the disk image so that it looks professional. Many developers now enable the user to drag the application icon onto an icon beside it on the disk image representing the Applications folder, so that the user doesn't have to open the Applications folder and drag it there directly. You can create a disk image yourself using Apple's Disk Utility application, but you may prefer to use one of several utilities that are available for the purpose that make it easier to set up the license presentation and other features. A good one is DropDMG, by Michael Tsai (who is this book's technical editor and the developer of the popular SpamSieve and EagleFiler applications). DropDMG is available at <http://c-command.com/dropdmg/>.

The theory behind the use of disk images is that it is easier and more obvious, not to mention more trustworthy, to install an application by dragging it to your Applications folder from a disk image than having to run an installer. The truth, I think, is more complicated. Installers are redolent of the Windows operating system, at least in the eyes of many Apple fans, and I have no doubt that Apple saw the disk image as yet another way to distinguish the Mac from PCs by providing greater ease of use. More recently, however, Apple has been voluble about recommending that you use installers instead of disk images when your application requires that supporting software be installed in different locations. The *Apple Human Interface Guidelines* tell you to "support drag-and-drop installation if your application bundle contains everything needed for the application to run" but to use an installer if files must be installed in specific locations or locations that require administrative access. The *Software Delivery Guide* concurs, although it proposes broader criteria for using a managed install. Although for a time Apple's *PackageMaker User Guide* took the opposite tack, a 2009

update has brought it into line. There have been suggestions that PC users switching to the Mac are mystified by disk images and the supposedly self-evident drag-install technique they embody. Perhaps Apple's recent friendliness toward installer applications is part of Apple's campaign to attract PC users to the Mac platform.

If you're going to distribute your application as shareware or commercial software, you have additional steps to take before you place it in a disk image or an installer package. You will have to consider a mechanism for generating and enforcing registration keys. You will probably find it useful to code into the application a free trial period as an exception to the registration key requirement. And you will of course need to create some mechanism to transfer your customers' money from their wallets to yours. None of this is easy.

I am aware of three major services that do most of this for you: eSellerate, part of Digital River, at <http://www.esellerate.net/>; Kagi, at <http://www.kagi.com/>; and FastSpring, at <http://www.fastspring.com/>. I am a happy user of eSellerate, but I have no reason to doubt that Kagi and FastSpring work as well. Explore all three, and any others you find, before making your decision.

Because I am a user of eSellerate's system, I will describe it. To use it, you sign up with the company and download the developer SDK it provides. You have to incorporate the SDK into your software, which requires writing a reasonably substantial amount of code yourself to make it accessible in a manner that fits in with your user interface. The SDK allows your users to run a mini-Web store inside your application, triggered, for example, by a menu item in your application to purchase a registration key generated by the company. The SDK also allows you to add registration tests when a user launches your application, so that your application can put up a dialog or respond in any other way you like if an unregistered copy is encountered. The company also offers you the ability to incorporate a Web store into your own Web site.

These systems take care of all the rest of the drudgery for you. They generate the registration keys; they handle credit card, PayPal, and other transactions; they handle foreign currency conversions; and they deposit your share of the proceeds in your bank periodically. They may also provide many other services. For example, you may be able to set up a system with the company to share your take with a partner or business associate and to arrange for a share to go to people who refer sales to you from their own Web sites.

For all of this, the companies of course take a share of the proceeds as their cut. The fee schedule depends on sales volume, typically with a break in favor of developers with low sales volume to encourage you to get into the game.

I find using one of these services to be an enormous benefit, even for my applications, which are highly specialized developer utilities that enjoy relatively low sales volume. The percentage that the company takes is entirely reasonable for the benefits

conferred. Purchases work with no participation required on my part other than to hold an occasional user's hand when the user can't figure out where the registration key is (answer: It was installed automatically by the software and the application is now already registered, plus it was reported to the user in the registration dialog and it was e-mailed to the user). My share of the proceeds shows up in my bank account automatically every month like clockwork. Because I elected to do so, I receive an e-mail message from the company every time a sale goes through, and I use a custom AppleScript to incorporate the report into a custom Excel spreadsheet. If I preferred not to track sales this closely myself, the company offers several other options for reports that I can receive or reports that I can look up on the company's Web site.

You could write code to do at least some of this, but I question why you would want to code all of it yourself. I doubt that you can handle the credit card, PayPal, and foreign currency transactions yourself, but PayPal, for example, offers an API for doing that. Of course, you could insist on receiving checks in the mail, but you would likely lose many sales if you did, and the time demand might prevent you from writing the next killer Mac OS X application.

Whatever system you use for managing sales transactions, be sure to offer a time-limited free trial version of your software. It's up to you whether you hobble it in some fashion until it is registered. Most users prefer a completely functional trial version so that they can realistically evaluate its ability to meet their needs. You won't have much difficulty coding a relatively foolproof method for disabling or crippling the application when the time period expires. Just make sure that you allow the user to start a new trial period when a new version of your application comes out. It is very frustrating for a potential customer to discover that the trial version of your latest release won't run because the user tried a much earlier version three years ago. Also, if you have the time, put in an early warning system that starts ticking down a few days before the free trial period expires. A user who knows the end is near is more likely to buy than a user who can no longer run the software. For good measure, I put a few surprise free bonus days in my applications, without advertising their presence, to give potential customers one last incentive to buy.

Don't spend a lot of time protecting your application against the pirates. Microsoft and Adobe have substantial financial incentives to prevent piracy, but unless your application sells in the thousands of units, you most likely do not. There are so many honest people in the world that you can more easily afford to let the pirates steal your application than you can afford the time it would take to try to stop them—and you wouldn't succeed in stopping them, anyway, because most of them are teenagers with more time on their hands than you.

Step 6: Promote the Application

Finally, there is the matter of marketing. They won't buy your application if they don't know it exists.

Version Tracker and MacUpdate are essential places to announce your product. Lots of Mac users watch those sites daily looking for the latest goodies and updates, and they search those sites when they're shopping for something in particular. But you should also invest a few hours in scouting out as many of the Web sites that are relevant to your product as you can. Since you wrote the application, you probably know most of them anyway, and there are a couple of dozen standard Mac sites that you should include in any list. Many of these sites list an e-mail address for press releases and similar announcements. Compose a press release and e-mail it to all of these sites. If you don't know what a press release looks like, find some and copy the format. The basic rule is to keep the first few lines really simple because that's all that some sites will publish, and keep the language clear, simple, and direct. Be sure there are some quotable quotes right after the first few lines, because the people who write up your product on their Web sites don't want to have to spend their valuable time composing prose of their own. Depending on the nature of your application and any reputation that precedes you, anywhere from a handful to all of the sites you contact will publish a note about your new release, together with a link to your Web site. The sales spike from an announcement on some of these sites will be obvious when you get your next report.

If you've written a special-purpose application appealing to a market that has a trade publication, consider buying an ad. Only you can decide whether the price is too steep, and there's no harm in asking. I have no idea what the price of an ad in MacTech Magazine, Macworld, or MacLife might be, but if you think your application has the potential to profit from it, ask.

You may receive inquiries from reviewers asking for a free NFR (Not For Resale) copy of your application. It probably doesn't pay to be overly suspicious. As I said before, the world is full of honest people, and if Joe Blow tells you he wants to review your application on his Web site, you should believe him (although you might want to check that his Web site exists). He might actually publish a review. And even if he doesn't, he probably wouldn't have paid for your application anyway. He is in the same category as the pirates: He is irrelevant if he steals your application, but very valuable if he reviews it—at least, if it's a good review, but that part is in your control because you wrote the software.

If your application is of interest to people who frequent a particular mailing list, you should frequent the list, too. Most people don't like to see blatant promotion of a commercial product on mailing lists, but there is a way to be subtle about it and provide value to the list. When a question comes up on the list on which you are an expert, and your application would solve the poster's problem, answer the question clearly and thoroughly, and drop a modest reference to your application and its download URL in the process. Nobody seems to mind that, because you're contributing.

Conclusion

I hope you have found this book to be useful and informative. I set out in this Second Edition to write up essential steps needed to create a real, working application, focusing on material that is not readily available in any comprehensive way in other books about writing Cocoa software. There are a dozen or more books in print about the details of using Objective-C and coding specific application features. But I am not aware of any other book that lays out in detail how a specific application was written and distributed from start to finish.

In Section 3, I will say just a little about where you can go from here.

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 13.

General Documentation

Xcode Project Management Guide (Building for Release)

Code Signing Guide

Technical Note TN2206: Mac OS X Code Signing In Depth

codesign(1) manual page

Xcode Unit Testing Guide

Apple Publications Style Guide

Apple Human Interface Guidelines (Software Installation and Software Updates)

Software Delivery Guide

PackageMaker User Guide

Looking Ahead

You have now built a working Vermont Recipes application with a fully functional Chef's Diary document. It includes all of the major trappings that go with any competent Mac OS X application, such as user preferences, printing, a Help book, and AppleScript support.

There is a lot of work ahead of you, however, if you choose to turn Vermont Recipes into a real recipes database application. Not least among the tasks remaining is the need to master Apple's Core Data API. And that is only for the data modeling and storage functionality that the application will require. To finish building out the recipes window, you will also need to use many AppKit classes that you have not encountered in this book, including NSOutlineView for the source list in the left pane of the recipes window, perhaps NSBrowser for a browser view in the top pane on the right side of the recipes window, maybe NSTableView for the bottom pane, and many kinds of controls for the recipes window's drawer. To add spice to the application, you will undoubtedly want to dive into Core Animation. For increased speed, you may want to explore Grand Central Dispatch and NSOperation. The possibilities are endless.

There isn't room in this book—or any other book—to cover all the possibilities, so I must let you go at this point. You will never be on your own, however, because there is a wealth of Apple documentation and third-party books to help you on your way. I hope that this book has given you a good start.

Before leaving you, in Recipe 14 I will touch upon a few Objective-C and Cocoa technologies that you should use in your own development work, but which I have not yet covered in this book. As I noted at the outset, virtually all Cocoa applications should make use of modern Objective-C and Cocoa technologies such as properties, Cocoa Bindings, and garbage collection. They greatly reduce the amount of work you must do as a developer, and they simplify your code. Properties usually make it unnecessary for you to write accessor methods, which tend to be boring and repetitive. Cocoa Bindings may not allow you to avoid much of the work you would otherwise do in Interface Builder to connect outlets to the data they represent, but they have the potential to reduce the amount of

controller codes you have to write. And garbage collection can relieve you of most of the tedium and potential for error involved with reference-counted memory management.

I deliberately avoided using these modern techniques through most of the book, not out of an old-fashioned nostalgia for ancient technology, but because you must understand how to use the older techniques even in the modern world. The newer technologies are built on top of the older ones, so you have to understand the older technologies in order to do effective debugging or optimization. In addition, there are circumstances where you can't use the newer technologies. For example, you will encounter many circumstances where properties can't be synthesized automatically because you must do additional work in your accessor methods. You may find it useful to set up complex connections in Interface Builder. And there are situations where garbage collection cannot be used.

Add New Technologies

With this recipe, you wrap up the book by converting one of the classes at the heart of the Vermont Recipes application so that, instead of using accessor methods, Interface Builder connections, and reference counted memory management, it uses properties, Cocoa Bindings, and garbage collection. You will make these changes in the `DiaryWindowController` class because it contains enough accessors, connections, and memory management code to demonstrate what is involved.

HIGHLIGHTS

Using properties

Using Cocoa Bindings

Using garbage collection

Apple advises you to use all of these newer technologies when you begin to develop a new Cocoa application, because they will save you a great deal of effort and produce much simpler and more manageable code. Apple advises you not to convert existing code to these technologies because, based on Apple's experience with its own conversion attempts, the effort does not yield a sufficient benefit to counterbalance the difficulty or the time required. In some cases, it might even require substantial refactoring of existing code.

I used the older techniques in this book because it remains essential to understand how they work, and sometimes you must still use them. I nevertheless undertake to convert one of the classes in this recipe, against Apple's advice, to show you what is involved.

Step 1: Switch to Properties

One of several new language features that Apple introduced with version 2.0 of the Objective-C language is known as *declared properties*. I won't discuss one aspect of properties, the dot syntax notation that you can use with it instead of Objective-C's traditional square bracket notation. But you should definitely declare properties instead of accessor methods, and let the computer synthesize them for you instead of implementing them in code yourself, wherever possible in your new Cocoa applications.

Typically, accessor methods are simple methods that you use to get or set the value of an instance variable. Using accessor methods instead of accessing an instance variable directly is desirable for a number of reasons. For example, it allows you to change the underlying means of storing the data in a class in a later release of an application without having to revise clients of the class. In addition, it allows you to consolidate memory management for an instance variable that is an object into a single location, the accessor methods, if the application uses reference counted memory management.

In the past, there were a number of commonly used ways to write getter and setter accessor methods. After some controversy a few years ago, Apple has settled on these models for the most common circumstances:

For a setter:

```
- (void)setMyArray:(NSArray *)array {
    if (myArray != array) {
        [myArray release];
        myArray = [array retain];
    }
}
```

This assumes that `myArray` is an instance variable declared in the class's header file. The method first checks to see whether the current object in the instance variable is the same object (that is, the same memory location) as the new array object being passed into the method. If so, there is no reason to do any work and it wouldn't be safe to release the array. Otherwise, the method releases the object currently referenced by the instance variable and then sets the instance variable to the new incoming object, retaining it in the process.

For a getter:

```
- (NSArray *)myArray {
    return [[myArray retain] autorelease];
}
```

Memory management is handled consistently in the two accessor methods. In a typical case, the expectation is that the existing value in the instance variable has been retained, possibly in the class's designated initializer or possibly as a result of a previous call to the same setter method.

The setter receives an object in its parameter. If the two objects are different (that is, they have different memory locations), the existing instance variable is released. This may call the object's `-dealloc` method if its retain count is reduced to 0. The instance variable's value is then replaced by the new object, which the method retains in order to comply with the expectation that instance variables are retained.

The getter does not simply return the retained instance variable. Instead, it retains it again and then balances the retain with an autorelease. This has the effect of extending the instance variable's life while leaving it in the same memory management state in the long term. A method that calls the getter can count on having enough time to assume ownership of it by retaining it to counteract the pending autorelease, even if, for example, some other thread releases the instance variable at the same time.

Apple recommends variations on this technique where, for example, speed is important and you know that the getter will be called more frequently than the setter, or if you know that extending an object's lifetime is not important. Also, you may have to call `-copy` or `-mutableCopy` instead of `-retain` in the setter in appropriate circumstances. More elaborate steps may have to be taken in a multithreaded application.

The advent of properties in Objective-C 2.0 has made it unnecessary to go to all that bother in most cases. Using properties saves you from the drudgery of writing accessor methods and the significant attendant risk of error. At the same time, it gives Cocoa greater power to dictate how getters and setters are designed, in the interest of improved reliability in Cocoa applications at large. When you declare a property and direct the compiler to synthesize its accessor methods for you, you delegate a certain degree of authority to Cocoa to make decisions on your behalf. Cocoa in exchange promises to do a good job of attending to all the considerations that bear on proper implementation of accessor methods. Properties do still leave you some ability to specify how the synthesized accessors work, however, by judicious use of attributes in the declaration.

The basic technique is to replace your accessor methods with an `@property` directive in the header file and an `@synthesize` directive in the implementation file of a class, protocol, or category. In 32-bit applications, you still have to declare an instance variable, but in 64-bit-only applications you don't.

I won't cover the details of when and how to use the various available attributes of properties here, because it is very well covered in the "Declared Properties" section of *The Objective-C Programming Language*.

1. Open the `DiaryWindowController.h` header file. Near the top, leave all the instance variable declarations as they are. However, delete the entire Accessor Methods section and replace it with this:

```
#pragma mark PROPERTIES
```

```
@property (readonly, nonatomic) IBOutlet NSTextView *diaryView;  
@property (readonly, nonatomic) IBOutlet NSTextView *otherDiaryView;  
@property (readonly, nonatomic) IBOutlet NSDatePicker *datePicker;  
@property (readonly, nonatomic) IBOutlet NSSearchField *searchField;
```

(code continues on next page)

```

@property (readonly, nonatomic) IBOutlet NSSplitView *splitView;
@property (readonly, nonatomic) IBOutlet NSButton *addTagButton;
@property (assign, nonatomic) id eventMonitor;
@property (assign, nonatomic) id didSaveDiaryDocumentObserver;
@property (assign, nonatomic) id didAutosaveDiaryDocumentObserver;
@property (assign, nonatomic) id didRestoreAutosavedDiaryDocumentObserver;
@property (assign, nonatomic) id restoreAutosavedDiaryDocumentAlert;
@property (assign, nonatomic) id userDefaultsDidChangeObserver;

```

You can think of each property declaration as being the same as declaring both accessor methods, or as declaring the getter method in the case of a read-only property. The rest of your code remains as it is, calling the getter and setter methods in the usual form as if they had been declared. Among other things, this keeps your code compliant with key-value coding.

The first several properties are declared **readonly** because there is no setter. The last several properties are declared using the **assign** attribute because, as you can see by looking at the implementation file, the setters perform a simple assignment operation rather than the more elaborate retain or copy operation described earlier. All of the properties are declared **nonatomic**, because we are not worried about thread issues here, and **nonatomic** is faster because it does not synthesize accessors that include thread safety code.

2. In the `DiaryWindowController.m` implementation file, delete the entire Accessor Methods section and replace it with this:

```

#pragma mark PROPERTIES

@synthesize diaryView;
@synthesize otherDiaryView;
@synthesize datePicker;
@synthesize searchField;
@synthesize splitView;
@synthesize addTagButton;
@synthesize eventMonitor;
@synthesize didSaveDiaryDocumentObserver;
@synthesize didAutosaveDiaryDocumentObserver;
@synthesize didRestoreAutosavedDiaryDocumentObserver;
@synthesize restoreAutosavedDiaryDocumentAlert;
@synthesize userDefaultsDidChangeObserver;

```

Again, it requires only one line to synthesize both the getter and setter accessor methods for each object.

Declaring a property does not require you to include an `@synthesize` directive in the implementation file. However, if you do not, you must write the two accessor methods, or the getter accessor method in the case of a read-only property. This flexibility allows you to use an `@property` declaration while writing custom accessor methods that perform more than the usual assignment or copy operation described above.

3. Build and run the application. Open the Chef's Diary window and add a bunch of diary entries with tags. Everything works as before.

Step 2: Switch to Cocoa Bindings

In Section 2, you spent a fair amount of time in Interface Builder connecting outlets and actions in code to view objects in nib files. In addition, you wrote a lot of what is called *glue code* to make sure that data is updated when the user manipulates a control and that the state of the control is updated to reflect changes in the data.

Cocoa Bindings are designed to help you eliminate much of the glue code. You still typically use Interface Builder, but instead of dragging connections using the Connections inspector, you use the Bindings inspector.

Apple recommends that you use Cocoa Bindings in every Cocoa application that you write from scratch. In addition, Apple advises that you can move an existing application to Cocoa Bindings, and that you can do it in stages over time. Apple offers your user preferences as one area that you can easily move to Cocoa Bindings all at once, using `NSUserDefaultsController`.

In most cases, Cocoa Bindings make use of prebuilt subclasses of `NSController`, which you instantiate in a nib file. In simple cases, you can instead bind directly to existing objects that you have already created, such as a window controller. Cocoa Bindings rely on key-value coding and key-value observing to coordinate the MVC model and view. Generally, you use `NSObjectController` for single objects or one of the collection controller subclasses of `NSController` for collections such as arrays and trees.

In this step, you once again focus on `DiaryWindowController`, converting some of its features to use Cocoa Bindings. Converting everything in the class, or in the application, to use bindings is a big task, and I'll leave that to you if you want to take it on. Here, I will just give two examples.

1. Start with the Add Entry button, and create a simple action binding for it.

Open the `DiaryWindow` nib file in Interface Builder.

Select the Add Entry button, and then open the Button Connections Inspector. In the Sent Actions section, click the little *x* in front of First Responder to break

its connection to the `-addTag:` action method. If you were to stop here, the Add Entry button would no longer work.

Open the Bindings inspector and expand the Target binding in the Action Invocation section. Select the checkbox at the top, and then choose File's Owner in the "Bind to" pop-up menu beside it. The File's Owner of the diary window nib file is `DiaryWindowController`. You are binding directly to `DiaryWindowController` without using an `NSObjectController` object.

In the Model Key Path combo box, use the menu to choose self.

In the Selector Name field, enter `addEntry:`. This is the selector for the `-addEntry:` action method.

Deselect the Conditionally Sets Enabled checkbox.

If you build and run the application now, the Add Entry button will work just as it did before. This example does not demonstrate any great benefit of Cocoa Bindings, because you have not eliminated any glue code or the need for the action method. You have only switched from using the Connections inspector to using the Bindings inspector to connect the action method to the user interface element that triggers it.

2. Next, set up a more complex binding.

Start by adding an `NSObjectController` instance to the `DiaryWindow` nib file. You use it in this step to update the date picker automatically so that it shows the date of the current diary entry, which is the entry containing the insertion point, every time the insertion point moves out of the current diary entry. You also use it to set the current entry automatically by moving the insertion point to a new diary entry based on any date that the user enters in the date picker. This replaces the `-updateDatePickerValue` method and the `-goToDatedEntry:` action method that you wrote in Recipe 4.

You also use the new `NSObjectController` instance to validate the date picker by enabling or disabling it automatically when the user adds or deletes a diary entry, depending on whether there are any diary entries in the document after the change. This allows you to remove the `ValidatedDiaryDatePicker` subclass of `NSDatePicker` that you created in Recipe 4 and to change the class of the date picker back to `NSDatePicker`.

First, delete the existing `-updateDatePickerValue` and `-goToDatedEntry:` methods in the `DiaryWindowController` header and implementation files. In addition, remove the call to the `-updateDatePickerValue` method in `DiaryWindowController`'s `-updateWindow` method, and remove the last clause of the `-validateUserInterfaceItem:` method in the `DiaryWindowController.m` implementation file. In addition, remove the `ValidatedDiaryDatePicker` class

from the `DiaryWindowController` header and implementation files. In Interface Builder, select the date picker in the `DiaryWindow` nib file's design surface, and in the Date Picker Identity inspector, use the Class pop-up menu to set its class back to `NSDatePicker`. What you're deleting is the old-style glue code that you don't need when you use Cocoa Bindings.

Because you deleted the action method, you should also delete its connection to the date picker. In Interface Builder, with the date picker in the design surface still selected, open the Validated Diary Date Picker Connections inspector. Click the little x before First Responder in the Sent Actions section to delete the connection.

Then, in Interface Builder's Library window, choose `Cocoa > Objects & Controllers > Bindings`, and drag an instance of `NSObjectController` into the nib file's document window. Rename it **Diary Controller**. This is an instantiated object controller object that will exist throughout the life of the diary window.

3. An object controller is an MVC controller. It needs two bindings, one to interact with its content—the MVC model—and the other to interact with the user interface elements that display and control the model's data—the MVC view.

To provide the new Diary Controller with a connection to its content, select the Diary Controller in the nib file's document window. Then, in the Object Controller Bindings inspector, disclose the Content Object binding in the Controller Content section. Select the checkbox at the top of the Content Object binding and choose File's Owner in the "Bind to" pop-up menu beside it. As you know, the File's Owner is the `DiaryWindowController`. Then enter this key path in the Model Key Path field: **self.document**. The diary document is now the content of the Diary Controller. Deselect the Conditionally Sets Editable checkbox.

Now enable the new Diary Controller to get or set the current diary entry. With the Diary Controller still selected, open the Object Controller Attributes inspector, click the Add (+) button at the bottom, and enter **currentEntryDate**. Repeat the process and enter **hasEntries**. These expose two model keys that you will use with the date picker. Note that the class of the Diary Controller is given in the Attributes inspector as `NSMutableDictionary` by default. Change it to **DiaryDocument**. The keys you just added refer to new methods that you are about to write in `DiaryDocument`.

Save the nib file, because it is all too easy to lose Interface Builder settings if you don't.

4. Write the **-currentEntryDate**, **-setCurrentEntryDate:**, and **-hasEntries** methods in the diary document. Cocoa Bindings will call these methods using key-value coding, based on the two keys you just added to the Diary Controller, whenever it needs to update the date picker's value or enabled setting and whenever it needs to select a new diary entry because the user changes the date

picker's value. Both getter and setter methods are required for the current entry date, but only a getter is needed for the `hasEntries` key. These three methods are very easy to write, because you've already written all the necessary supporting methods, and you already worked out the logic in the `-updateDatePickerValue` and `-goToDatedEntry:` methods you just deleted.

Add these declarations at the end of the `DiaryDocument.h` header file:

```
#pragma mark BINDINGS SUPPORT

- (void)setCurrentEntryDate:(NSDate *)date;
- (NSDate *)currentEntryDate;

- (BOOL)hasEntries;
```

Implement the methods at the end of the `DiaryDocument.m` implementation file:

```
#pragma mark BINDINGS SUPPORT

- (void)setCurrentEntryDate:(NSDate *)date {
    NSTextView *keyView =
        [[[self windowControllers] objectAtIndex:0] keyDiaryView];
    NSRange targetRange = [self firstEntryTitleRangeAtOrAfterDate:date];
    if (targetRange.location != NSNotFound) {
        [keyView scrollRangeToVisible:targetRange];
        [keyView setSelectedRange:targetRange];
    }
}

- (NSDate *)currentEntryDate {
    NSUInteger insertionPointIndex = [[[self windowControllers]
        objectAtIndex:0] insertionPointIndex];
    NSRange range =
        [self currentEntryTitleRangeForIndex:insertionPointIndex];
    if (range.location == NSNotFound) {
        return [NSDate date];
    } else {
        return [self dateFromEntryTitleRange:range];
    }
}

- (BOOL)hasEntries {
    return [self firstEntryTitleRange].location != NSNotFound;
}
```

The first two methods are modeled very closely on the `-updateDatePickerValue` and `-goToDateEntry`: methods you originally wrote in the `DiaryWindowController` class in Recipe 4. These methods are not typical getters and setters because they don't front for an instance variable, but there are no hard and fast rules for writing getters and setters and Cocoa Bindings won't know the difference.

Placing these methods in the `DiaryDocument` class may not seem entirely appropriate from an MVC point of view, because the current diary date is not stored as a model data value in the document but is instead defined in terms of the current location of the insertion point in the view. Nevertheless, conceptually it may be considered a model value, and it is useful to do so here to illustrate the use of Cocoa Bindings. This step would work as well if you placed these methods in `DiaryWindowController`, but you would have to make minor changes from what is described in the next several paragraphs.

5. Now you can bind the date picker to the Diary Controller. As a result, the date picker will always reflect the date of the current diary entry—that is, the diary entry containing the insertion point—and the current diary entry will always reflect the date in the date picker, even when the user moves the insertion point. Furthermore, when the user changes the date in the date picker, the insertion point will move to the new current diary entry. Finally, the date picker will be enabled and disabled if the user does something to create the first diary entry or delete the last diary entry.

Select the date picker control in the design surface and open the Date Picker Bindings inspector. Note that this is no longer the Validated Diary Date Picker Bindings inspector because you just changed its class back to `NSDatePicker`.

Then disclose the Value binding. Select the Bind checkbox at the top and choose Diary Controller in the “Bind to” pop-up menu beside it. Enter **content** in the Controller Key combo box, and choose one of your new keys, `currentEntryDate`, in the Model Key Path combo box's pop-up menu. Deselect the Allows Editing Multiple Values Selection checkbox and the Conditionally Sets Enabled checkbox.

Perform a similar operation on the Enabled binding in the Availability section. Disclose it, select the checkbox, choose Diary Controller from the “Bind to” pop-up menu, and enter **content** in the Controller Key combo box. This time, choose `hasEntries` in the Model Key Path combo box's pop-up menu.

Now the Diary Controller is bound on both ends. It knows that its content is in the diary document, and it knows how to interact with the date picker in the diary window.

Save the nib file to preserve the changes.

6. In most circumstances, bindings would now work automatically without more. Here, however, you must take one additional step to force the Diary Controller to send KVO notifications to the date picker when the user moves the insertion

point in the diary window. Specifically, when the user moves the insertion point into a different diary entry, the Diary Controller will not notice the change unless you tell it.

The circumstances under which you must trigger KVO notifications explicitly tend to confuse newcomers to Cocoa Bindings, and this accounts for a lot of traffic on the developer mailing lists. Basically, if you designate a property or accessor as a key and bind the object controller to the object where the property or accessor is located, Cocoa Bindings works automatically. Here, however, the Diary Controller knows nothing about the navigation buttons that move the insertion point, and even less about what the user does with the mouse button or the arrow keys. When the user employs any of these techniques to move the insertion point, you must inform the Diary Controller about it.

To do this, bracket the single statement currently in the `-windowDidUpdate:` delegate method with calls to `-willChangeValueForKey:` and `-didChangeValueForKey:` for each of the two keys that the Diary Controller is observing, `currentEntryDate` and `hasEntries`. Revise the delegate method so that it looks like this in its entirety:

```
- (void)windowDidUpdate:(NSNotification *)notification {
    [[self document] willChangeValueForKey:@"currentEntryDate"];
    [[self document] willChangeValueForKey:@"hasEntries"];
    [self updateWindow];
    [[self document] didChangeValueForKey:@"hasEntries"];
    [[self document] didChangeValueForKey:@"currentEntryDate"];
}
```

Normally, you would place the calls to `-willChangeValueForKey:` and `-didChangeValueForKey:` in a setter accessor method that changes a value that the object controller cares about or, in this case, in the action methods that change the insertion point. But then the notification would not be sent when the user clicks in a new diary entry in the diary window instead of using one of the navigation controls at the bottom of the window. By putting the calls in the `-windowDidUpdate:` delegate method, you ensure that changes to the insertion point made by clicking in the window or using the arrow keys are also noticed by the Diary Controller.

You make these calls in `DiaryWindowController` rather than `DiaryDocument` because it is `DiaryWindowController` that knows when the insertion point moves. It doesn't matter to Cocoa Bindings where the methods that are triggered are located, as long as the Diary Controller is bound to the object where they are located and the methods are designated as its keys.

7. Build and run the application, and test the date picker. If you create a new diary document having no diary entries, you see that the date picker holds the current date and it is disabled. When you click Add Entry, the date picker immediately

becomes enabled, and its date changes to the date in the new entry's title. Add a few other entries, and the same thing happens every time. Now use the navigation buttons to move the insertion point from one entry to another, and watch the date in the date picker change to match. You can even add a tag to one of the entries, and then when you use the search field to find it, the date in the date picker changes to match the newly selected diary entry. Move the insertion point to another diary entry by clicking with the mouse or using the arrow keys on the keyboard, and the date picker's date changes to match. Finally, enter a new date in the date picker, and you will see the indicated diary entry selected immediately if the new date matches the selection criteria you established earlier.

All of this works because of the Cocoa Bindings you have set up. You were able to remove a lot of glue code.

Step 3: Switch to Garbage Collection

Garbage collection came to Cocoa in Leopard, as an alternative to the traditional reference counted memory management system that relies on explicit `-retain`, `-release`, and `-autorelease` calls. In many applications, turning on garbage collection should work with almost no effort on your part, but there are lots of exceptions and special rules that require special coding. There is no substitute for reading Apple's documentation to make sure you get all the nuances correct.

Once you've worked with the traditional reference counted memory management system for a while, it becomes second nature. It has always been a bit difficult for beginners, but only until they've spent some time using it. With even a little experience, it hardly requires any thought at all to get it right. Some naysayers would have it that garbage collection isn't needed.

Nevertheless, the siren song of garbage collection—that it makes memory management automatic—is hard to resist. There are even those who say that garbage collection is rapidly becoming mandatory, especially if you need to use plug-ins or frameworks that require it.

Turning on garbage collection is simple. You only have to set a single build setting. You have three choices: no garbage collection, garbage collection with reference counted memory management side by side, and garbage collection required. For a new Cocoa application project, requiring garbage collection is a reasonable choice. Most of the time, your code will be simpler to write, requiring no calls to `-retain`, `-release`, or `-autorelease`.

Even when you're updating an existing application, however, you may be well advised to elect garbage collection required. One of Apple's nice touches was to

allow garbage collection to be turned on even though your existing application is riddled with `-retain`, `-release`, and `-autorelease` calls. Those calls are simply ignored when garbage collection is turned on.

As an avid AppleScript fan, I must warn you that a serious bug in Mac OS X 10.5 Leopard makes AppleScript pretty much unusable in a garbage-collected application. In Vermont Recipes, therefore, turn on garbage collection only in the Snow Leopard target.

To turn on garbage collection, set the `GCC_ENABLE_OBJC_GC` (Objective-C Garbage Collection) field in the GCC 4.2 – Code Generation section in the Build pane of the Vermont Recipes SL target’s Info window. The choices available to you are Unsupported, Supported, and Required. The default is Unsupported. The Supported setting is primarily for frameworks and libraries that can be included in many applications and must therefore provide complete support for both reference counted memory management and garbage collection. For the Snow Leopard version of Vermont Recipes, choose Required from the pop-up menu.

In a garbage-collected application, the `-dealloc` method is ignored because it is called only when a reference counted application object’s retain count becomes 0. If a garbage-collected application needs to perform other kinds of cleanup just before it goes away, you should implement the `-finalize` method instead. Implementing a `-finalize` method is discouraged if you can find a way to perform necessary cleanup at some other point in an object’s life cycle. This is because garbage collection performs its memory management obligations periodically in bunches. Your `-finalize` methods may be called all at once, resulting in noticeable pauses in application execution from time to time, and they may be called long after you wanted to get rid of resources that are no longer needed.

Search Vermont Recipes for all of its `-dealloc` methods, to see whether any of them do anything other than release objects or call a superclass’s implementation that does anything else. It turns out that there are five `-dealloc` methods in Vermont Recipes, and only two of them do something more than release objects. The `-dealloc` methods in `DiaryPrintPanelAccessoryController` and `PreferencesWindowController` remove observers of notification center notifications. Fortunately, in a garbage-collected environment you do not have to remove notification center observers. So just leave these `-dealloc` methods as they are—they won’t be called—or delete them without moving the `-removeObserver:` calls somewhere else. In Snow Leopard, you no longer have to remove KVO observers in a garbage-collected environment, either. Since you are turning on garbage collection only in the Vermont Recipes Snow Leopard target, you therefore don’t have to implement any `-finalize` methods for that. Besides, you removed the two KVO observers in `DiaryWindowController`’s `-windowWillClose:` method, anyway, not in its `-dealloc` method. You therefore don’t have to implement any `-finalize` methods at all.

You don't have to do anything else, either. Vermont Recipes does not at this point incorporate any of the features that require special handling in a garbage-collected application.

You can, if you wish, go through the entire application now and remove all of the existing calls to `-retain`, `-release`, and `-autorelease`, but you don't have to because they will be ignored.

Conclusion

Remember that the next great Mac OS X application has not yet been written. It might not even have been conceived of yet. I say this to remind you that you might be the developer who thinks it up and writes it—and who reaps the rewards. But it won't happen if you don't try, so get to work! Thinking up new ideas isn't easy, but we all have it in us. Even if it is only a clever improvement on an old idea, there will be a market for it if it is good enough.

I should leave you with this warning, however: I don't think it will be a recipes application. You should always research your competition before you settle on a new project. Here is what I found when I researched Mac OS X recipes applications. There are an awful lot of them out there already, and some of them are very, very good. I list them here in alphabetical order. My apologies to the developers of any I missed.

- BusyCooks RecipeDB, from ToThePoint Software, at <http://www.busycooks.com/>
- The Computer Cookbook, from Craig Rhodes, at <http://www.papax2.com/>
- A Cook's Books, from 3 Cats and a Mac, at <http://www.3caam.com/products.html>
- Computer Cuisine Deluxe, from inaka software, at <http://www.inakasoftware.com/cuisine/index.html>
- Connoisseur, from Concept Development, at <http://www.connoisseurx.com/>
- Cookware Deluxe, from DigitalFriedChicken, at <http://www.digitalfriedchicken.com/CookWare.html>
- iCuistot, from Cafederic, at <http://www.cafederic.com/en/overview.html>
- Kitchen, from epigroove, at <http://www.epigroove.com/kitchen/>
- Shop'NCook (several applications), from Rufenacht Innovative, at <http://www.shopncook.com/>
- MacGourmet, from Advenio, at <http://www.advenio.com/macgourmet/>

- MacGourmet Deluxe, from Mariner Software, at <http://www.marinersoftware.com/sitepage.php?page=130>
- Measuring Cup, from Shallot Patch, at <http://www.shallotpatch.com/MeasuringCup/index.html>
- myRecipes, from MOApp Software Manufactory, at <http://createlivelove.com/applications/myrecipes/myrecipes.html>
- The Recipe Box, from Sonora Graphics, at <http://www.sonoragraphics.com/recipebox.html>
- SousChef, from Acacia Tree Software, at <http://acaciatreesoftware.com/>
- Yum, from Dare to Be Creative, at <http://creativebe.com/yum/>
- YummySoup!, from HungrySeacow Software, at <http://hungryseacow.com/>

There is even a Recipe Markup Language—based on XML, of course—known as RecipeML (formerly DESSERT), at <http://www.formatdata.com/recipeмл/>.

And don't overlook Apple's Core Recipes sample code at <http://developer.apple.com/mac/library/samplecode/CoreRecipes/index.html>. Apple describes it as “a series of projects to manage and manipulate recipe information using Core Data and Cocoa Bindings.”

DOCUMENTATION

Read the following documentation regarding topics covered in Recipe 14.

General Documentation

The Objective-C Programming Language (Declared Properties)

Cocoa Bindings Programming Topics

Cocoa Application Tutorial Using Bindings

Key-Value Coding Programming Guide

Key-Value Observing Programming Guide

Garbage Collection Programming Guide

Value Transformer Programming Guide

Memory Management Programming Guide for Cocoa

Index

%@ placeholder, using, 149, 155
() (parentheses), using with `if` statement, 104
= (assignment operator), using with `-init` method, 103–104

A

abstracts, adding to search experience, 506–509
accessibility features. *See also* VoiceOver utility
 adding, 341–345
 adding attributes, 339–340
 adding titles for date picker, 343
 connecting title attributes, 344
 custom controls, 338
 help field, 338
 help options, 341
 hierarchy of, 340
 introduction of, 337–338
 navigation buttons, 341–345
 rules for providing descriptions, 338–339
 supplying link attribute, 339
 testing descriptions, 340
 testing help attributes, 340
 title attribute, 343
 turning on VoiceOver feature, 340
accessor methods. *See also* methods
 declaring for Add Tag button, 325
 memory management, 600
 model-controller in, 126
 using, 600
 using with autosave operations, 290–291
 using in text systems, 124–125
 writing for accessory view controller, 377
accessory view controller
 accessor methods, 377–378
 action methods, 379
 adding to Print panel, 381–389
 connecting action methods, 379
 creating in Xcode, 374–381
 factory method, 375–376
 invoking setter methods, 379
 print info settings, 377–378
 print job code, 382
 represented object, 378–379
 testing, 384–386
action messages, considering responder chain for, 198–199
action methods. *See also* methods
 resource for, 85
 signatures of, 148
 verifying connection of, 148
 writing body of, 148
actions and outlets, using with Find submenu, 204–205
Add Entry button
 behavior of, 149
 comparing to Add Tag push button, 160
 testing, 156–157, 159, 167–168
 using Debugger Console with, 148
Add Entry menu item, enabling, 212
Add Tag button
 code reuse, 161
 comparing to Add Entry button, 160
 considering MVC design pattern, 161
 creating outlet for, 323–324
 declaring accessor methods, 325
 declaring protocols, 177
 DiaryDocument methods, 161
 features of, 160–161
 forcing identity to change, 328–329
 installing event monitor for, 324–327
 locating entry marker, 162–163
 location of insertion point, 162–163
 managing edge cases, 162
 obtaining range of entry titles, 162
 obtaining range of tag titles, 162
 placement of navigation code, 161
 positioning title tag, 164–165
 running and testing, 177
 scrolling text view, 167
 testing, 167–168
 use of NSRange struct, 161
Add Tag menu item, allowing changes to, 316–321.
 See also dynamic Add tag
-addEntry: action method
 adding stub of, 148
 calling, 149
 considering MVC design pattern, 149
 obtaining text storage object, 151
 Redo menu item, 152
 sending notifications related to, 152
 Undo menu item, 152
 writing body of, 149–152
-addTag: action method
 code structure of, 160
 using with Add Tag push button, 166–167
alerts
 adding help buttons to, 509–510
 creating, 294–295
 creating callback method for, 296
alias file, using, 16
alias records and files, resolving, 227
aliases versus bookmarks, 228
AppKit, built-in hooks, 77
Apple Help. *See also* help book
 advanced features, 510–511
 advantages and disadvantages, 484–485
 HelpViewer application, 485
 implementing, 484–486

- AppleScript language, 530
 - creating sdef file, 522
 - creating terminology dictionary, 520–525
 - decrypt object-first command, 578–580
 - encrypt object-first command, 578–580
 - features of, 519–520
 - implementing terminology, 531
 - sort verb-first command, 574–578
 - Standard Suite, 524
 - terminology, 520–522
 - terminology suites, 525
 - terminology version property, 531
 - to-one relationships, 530, 545
 - turning on support, 523
- AppleScript link, adding to topic page, 502–503
- application delegate class, using with main menu, 194
- application icons, adding, 353–356
- application settings, controlling globally, 449
- applications. *See also* Vermont Recipes application
 - associating documents with, 38–39
 - creator code, 41
 - packaging for delivery, 7
 - signature, 41
- assignment operator (=), using with `-init` method, 103–104
- autosaved documents. *See also* documents
 - restoring, 287, 292–294
 - testing, 296–297
- autosaving, turning on, 285–286
- Autosaving section, implementing, 474–477
- autosizing behavior, setting for split views, 107–108

B

- Backup.vrdiary
 - attempting opening of, 247
 - generating error alert for, 242
 - saving, 241
- binding, switching to, 603–609
- blocks feature
 - overview of, 321–323
 - using with dynamic buttons, 320
- bookmarks versus aliases, 228
- build folder
 - displaying in Finder project window, 16
 - managing, 19
- Building pane, using in Xcode, 19
- bundle identifier, indicating, 117
- button images, requirements for, 144
- Button Size inspector, using, 141
- buttons
 - default behavior of, 141
 - editing titles of, 141
 - enabling and disabling, 168
 - fixing autosizing behavior of, 141
 - renaming, 141

C

- C language
 - C99 dialect, 156
 - Objective-C as superset of, 3–4
 - resource for, 4
- %C placeholder, using, 155–156

- C99 dialect, using with C language, 156
- categories
 - declaring informal protocols, 218
 - NSServicesRequests, 217
 - using with NSScreen class, 262–263
- centering, turning off in printing, 412
- Character Viewer, adding to menu bar, 156
- checkboxes, using with accessory views, 372–373
- Chef's Diary, 107–108. *See also* current diary
 - document; diary window
 - adding preferences to, 442–443
 - applying RTF formatting capabilities, 107
 - building and running, 120–121
 - completion of, 191
 - controls in Printing section of, 470
 - creating New menu item for, 114–115
 - creating VRDocumentController class, 111–112
 - creating Window Controllers subgroup, 92
 - described, 90
 - DiaryDocument class, 91–94
 - displaying user default printing settings, 471–472
 - document data, 90
 - dragging horizontal line object, 443
 - formatted text in, 121
 - implementing controls, 470
 - printing customizations, 369–370
 - saving snapshot of, 94–96
 - selecting, 443
 - setting current document, 477
 - testing Autosaving section, 481
 - testing Document section, 481
 - testing window section, 480–481
- Chef's Diary document, opening, 112–114
- Chef's Diary tab view item. *See* tab view item
- Chef's Diary window, purpose of, 121
- class extensions, relationship to categories, 218
- class name, declaring, 22–23
- Class Reference document, consulting, 137–138
- classes
 - adding methods to, 216–218
 - checking header files for, 138
 - choosing in Interface Builder 3.2, 64
 - extending functionality via categories, 218
 - relationship to protocols, 169
 - using categories with, 217
- Classes group
 - creating subgroups in, 92–93
 - displaying in Xcode project window, 17
- Coco views, resources for, 63
- Cocoa
 - action messages, 86
 - first responder object, 86
 - responder chain, 86
 - versioning mechanism, 77
- Cocoa AppKit, built-in hooks, 73
- Cocoa Bindings, switching to, 603–609
- Cocoa design patterns, resource for, 4–5
- Cocoa frameworks
 - capabilities of, 6
 - design patterns, 6
 - resource for, 6
- Cocoa functionality, providing “for free,” 5–6

- Cocoa methods, getting help with, 74
- Cocoa Simulator
 - testing drawer in, 85
 - using in Interface Builder 3.2, 65–66
- code, labeling sections in classes, 219–221
- code completion, turning off, 19
- Code Sense pane, function of, 19
- comments versus `#pragma mark` statements, 216, 221
- controller, role in MVC design pattern, 30, 126
- controls
 - checking positioning of, 145
 - determining selection of, 146
- copying user interface elements, 142
- copyright notice
 - changing, 22
 - including in Info.plist file, 41–42
- Core Data, resource for, 12
- creator code
 - registering, 41
 - use of, 41
- Credits.rtf file, editing, 33–35
- curly quotes, using in error strings, 250
- current diary document. *See also* Chef's Diary; diary document; document behavior
 - assigning action to menu item, 234
 - assigning title to menu item, 234
 - backing up, 240
 - converting alias record to file URL, 228
 - converting bookmark data to file URL, 228
 - converting URL to NSData object, 225–226
 - creating alias record, 225–226
 - creating bookmark, 225–226
 - enabling and disabling menu item, 235
 - enabling menu item for, 225
 - file handling by NSURL class, 227
 - implementing protocol method for, 224
 - iterating over list of open documents, 233–234
 - reading and writing URL in user defaults, 232
 - requirements for, 222–223
 - testing, 239–240
 - testing save operation, 231
 - user interface features, 222
 - VRDocumentController class, 224
 - writing NSData object to user defaults, 231

D

- data model, controlling, 21
- data-bearing objects, copying versus retaining, 125–126. *See also* objects
- date picker
 - binding to Diary Controller, 607
 - connecting, 186
 - declaring ValidateDiaryDatePicker subclass, 183–184
 - determining display date, 184–185
 - dragging to diary window, 141–142
 - NSDatePicker class, 181
 - running and testing, 186
 - as title for current entry, 180
 - updating displayed value, 185
 - verifying lack of keyboard focus, 185
- date string, creating, 154–155

- debugger console window
 - using with Add Entry push button, 148–149
 - using with Find submenu, 203
- declarations
 - searching in Xcode, 48
 - showing in Editor Function pop-up menu, 19
- declared properties, using, 599–603
- decrypt object-first command, adding, 578–580
- default document name, providing, 346–349
- delegate argument, using, 159
- delegate methods. *See also* notifications
 - function in window controller, 30–31
 - notification object parameter, 135
 - NSSplitView, 75
 - preventing duplicate code, 76–77
 - use of *should* with, 75
 - using as hooks in Interface Builder, 73
- delegation versus notifications, 277
- delete command, supporting for diary entries, 573–574
- design patterns. *See also* MVC design pattern
 - callback selector to temporary delegate, 159
 - Target-Action, 86
- designated initializer, explained, 102
- Developer folder
 - creating link to, 11–12
 - location of, 11
- dialogs, adding help buttons to, 509–510
- Diary Controller, binding date picker to, 607
- diary document. *See also* current diary document
 - adding Info.plist file, 115–121
 - autosaving, 287
 - backing up, 297–298
 - getting and setting text of, 555–556
 - managing nib files in, 94
 - printing, 369
 - scripting as text, 542–544
 - sizing document windows, 260
 - storing and retrieving, 132–133
 - typing file path for, 477–480
- diary document name, providing default for, 345–349
- diary entries
 - CreateCommand class, 572
 - delete command for, 573–574
 - implementation of make command, 572–573
 - implementing NSKeyValueCoding, 568–569
 - marking beginning of, 155
 - ranges of, 547
 - restricting, 152
 - subclassing NSCreateCommand, 570–572
 - using KVC-compliant method with, 564–565
- diary entry class, 548
 - adding class element to sdef file, 545–546
 - creating, 538–541
 - declaring getter accessor methods, 557–558
 - declaring setter methods, 559–563
 - implementing make command for, 566–568
 - reasons for creation of, 555–556
 - to-many relationship, 537
 - using SLOG macro with, 541
 - writing -date method, 559
 - writing KVC methods, 550

- diary entry objects, invalidating array of, 548
- diary entry property, adding to document class, 563–564
- Diary menu
 - adding to menu bar, 200–201
 - enabling menu items in, 201–202
 - validating menu items for, 201–202
- Diary Tag Search menu item
 - building and running application, 205
 - completing, 204–205
 - creating behavior for, 205
 - eliminating validation of, 206
 - fixing `DiaryWindowController` class, 208
 - fixing `VRDocumentController` class, 208
 - function of, 204
 - implementing responder chain, 205–206
 - testing diary window, 207
 - using, 213
- diary text, storing, 123
- diary window. *See also* Chef's Diary; document windows; recipes window; windows
 - adding scrolling text views to, 104–108
 - archiving, 136
 - autosaving position of divider in, 282–284
 - autosizing split view, 107–108
 - building and running, 136
 - configuring split view, 133–135
 - dragging date picker to, 141–142
 - dragging Push Buttons to, 140–141
 - dragging search field into, 142
 - dragging square button to, 142
 - opening, 132
 - placing code for custom behavior of, 268–269
 - positioning in center of screen, 268
 - reducing cluttering in, 146
 - resizing in Interface Builder, 145–146
 - saving, 136
 - setting autosave name for, 276
 - setting initial size of, 268
 - setting minimum width of, 145–146
 - user's expectations of, 146
- diary window frame, autosaving and restoring, 280–282
- `DiaryDocument` class
 - creating in Xcode, 91–94
 - formatting date and time in, 154
- `DiaryEntry` class, creating, 553–555
- `DiaryWindow` nib file
 - opening in Interface Builder, 104
 - resizing Horizontal Split View, 104
- `DiaryWindowController` class
 - connecting `delegate` outlet, 99
 - connecting `window` outlet, 99
 - creating, 97–98
 - implementation of `-init` method, 100–104
 - initializing window controller, 100–104
 - making connections, 99
 - managing nib file, 99
 - placing action methods for Diary menu in, 201
- dictionary. *See* terminology dictionary
- disk images, using for distribution, 592–593
- display name, internationalizing, 351–352
- display screen, returning, 264
- display size, setting, 262
- divider, autosaving position of, 282–284
- document, behavior of, 30
- document attributes, storing for text system, 132
- document behavior. *See also* current diary document
 - applying current diary document, 221–223
 - archiving project, 257
 - building and running application, 257
 - categories, 216–218
 - saving project, 257
- document class, adding diary entry property to, 563–564
- document data, defining custom format for, 130
- document icons, adding, 353–356
- document methods, using, 26–28
- document subclass
 - function of, 21
 - writing, 60
- document text property
 - adding, 542–544
 - treating as implied container, 544
- document types
 - owning, 116–121
 - requirements for, 37
- document windows. *See also* diary window; recipes window; window controller; windows
 - attaching drawer to, 81
 - autosaving position and size of, 274–276, 278–282
 - position of parent window on screen, 265
 - setting initial position and size, 267–269
 - setting size of, 260–261
 - situations for opening, 127
 - user versus standard states of, 269
 - using resize control for, 261
 - zooming, 269
- documentation
 - consulting, 138, 192
 - for Interface Builder, 88
 - for main menu, 214
 - overview documents, 138
 - programming guides, 138
 - protocol documents, 306
 - providing for application, 589–590
 - release notes, 138
 - sample code, 584
 - third-party commentary, 366
 - Xcode, 52
- Documentation pane, using in Xcode, 20
- document-based applications
 - editing `Credits.rtf` file, 33–35
 - MVC design pattern, 21
 - naming templates for, 22
 - revising header files, 21–23
 - revising implementation files, 21–23
- document-based applications, creating, 12
- documents. *See also* autosaved documents; saved documents
 - associating with applications, 38–39
 - opening, 237–239
 - subclasses of, 20

- Documents folder, creating subfolders in, 14
- drawer. *See also* Recipe window drawer
 - attaching to document window, 80
 - examining connections for, 81
 - explained, 80
 - testing in Cocoa Simulator, 85
- Drawer Attributes inspector, using, 82
- Drawer Connections inspector, using, 82
- Drawer Content View object, using, 82
- Drawer object, hooking up, 82
- Drawer Size inspector, using, 82, 265
- dynamic Add tag. *See also* Add Tag menu item
 - using with Tag All button, 320, 323–330
 - using with Tag All menu item, 316–319

E

- echo command, text related to, 256
- editing files, 21–22
- Editor Function pop-up menu, showing
 - declarations in, 19
- element, connotations of, 545
- encrypt object-first command, adding, 578–580
- English.lproj folder, 15–16
- error alert
 - constructing strings for, 250
 - opening, 242
 - preventing, 241
- error code
 - changing, 241
 - using, 242
- error description, improving, 246–248
- error domains
 - changing, 241
 - NSErrorDomain, 242
- error handling code
 - clarifying failure reason in, 248
 - FIXME: comment, 233, 252–253
 - hierarchy of, 245
 - NSErrorRecoveryAttempting informal protocol, 250–251
 - testing, 241–242
 - using defined terms for strings, 251
- error local variable, declaring, 253–254
- error messages, improving, 242–244
- error objects
 - adding information to, 248
 - constructing and displaying, 245
- error parameters, passing NULL to, 253
- error strings, using curly quotes in, 250
- errorInfo dictionary, constructing, 244
- escape sequence, using for Unicode characters, 156
- eSellerate's system, using, 593
- event monitors
 - installing for Add Tag button, 324–327
 - removing, 327
- eventMonitor
 - using with blocks and notifications, 331–332
 - using with Tag All button, 325
- events
 - monitoring and responding to, 326–327
 - monitoring relative to autosaving, 286

F

- factory method, writing for accessory view controller, 375–376
- file path, typing for diary document, 477–480
- file URL
 - converting alias record data to, 228
 - converting bookmark data to, 228
 - identifying, 229–230
- filenames, changing in Xcode projects, 24–25
- files. *See also* renamed files
 - editing in separate windows, 22
 - verifying in Trash, 236
- File's Owner proxy
 - setting up in Interface Builder, 97
 - using in Interface Builder, 60
- Find command, using, 186
- Find submenu
 - creating keyboard shortcut for, 203
 - creating menu item for, 203
 - outlets and actions, 204–205
 - using debugger console window, 203
- Finder, changing filenames in accidentally, 24–25
- Finder project window
 - build folder in, 16
 - code files in, 16
 - folders in, 15–16
 - Importer folder in, 16
- Finder versus Xcode project window, 14–15
- first responder object. *See also* objects
 - configuring for preferences window, 444
 - using with Read Me item for Help menu, 198–200
- floor() function, using, 77
- folders
 - accessing for Vermont Recipes, 14
 - creating for Vermont Recipes, 14
 - creating in Xcode, 14
 - in Finder project window, 15–16
- fonts, obtaining, 417
- footer method, overriding, 416–417
- footers
 - calculating vertical position of, 419
 - defining left tab stop, 418
 - installing tab stops, 417
 - printing customizations of, 415–421
 - replacing default fonts, 417
 - repositioning for print scaling, 428–430
 - turning on, 416

G

- garbage collection, turning on, 603–604
- General pane
 - changing checkboxes in, 456–457
 - checkboxes in, 453
 - updating checkboxes in, 456
- General tab view item
 - alert for limit of print scaling, 458
 - alert for reopening autosaved document, 459
 - checkboxes in, 449, 452–453
 - declaring getter accessor methods, 450
 - declaring instance variables for checkboxes, 450

- General tab view item (*continued*)
 - registering notifications for checkboxes, 453–455
 - removing observer, 455
 - writing action methods for accessor methods, 450–452
- genstrings command-line tool, using, 46, 255–256
- getter accessor methods, writing, 600–601
- getter and setter, using *is* in, 291
- getter method
 - declaring for instance variable, 127
 - using in text system, 125
- global application settings, controlling, 449
- glue code, eliminating via Cocoa Bindings, 603
- graphic images. *See also* images
 - adding to projects, 144–145
 - obtaining, 143–144, 353
- groups
 - creating groups within, 145
 - creating subgroups of, 92–93
- Groups & Files pane
 - appearance of, 14
 - expanding, 18

H

- .h file extension, using with header files, 23
- header files
 - checking for classes, 138
 - organizing, 18
 - revising, 21–23
- header method, overriding, 416–417
- headers
 - calculating vertical position of, 419
 - defining left tab stop, 418
 - installing tab stops, 417
 - printing customizations of, 415–421
 - replacing default fonts, 417
 - repositioning for print scaling, 428–430
 - setting and drawing vertical origin, 420–421
 - turning on, 416
- help book. *See also* Apple Help
 - adding AppleScript link to topic page, 502–503
 - adding navigation pages, 496–501
 - adding task pages, 496–501
 - adding title page, 496–498
 - adding topic pages, 498–501
 - building and running application, 501–502
 - creating, 489–490
 - creating title page for, 490–491
 - implementing for Leopard and earlier, 511–517
- Indexer utility, 494
- lines in title page, 491–492
- meta tags, 492
- populating English.lproj subfolder, 490
- registering, 495
- resizing icons in, 498
- revising title page, 496
- help book files
 - emulating, 486–487
 - localizations of, 485
 - title page, 488

- help bundle, location of, 487
- help buttons
 - adding to alerts, 509–510
 - adding to dialogs, 509–510
 - adding to panels, 509–510
- Help menu, adding Read Me item to, 195–198, 200. *See also* Apple Help
- help:* protocol, using in HelpViewer, HelpViewer application, 503–506
- help tags, adding, 334–337
- Help.html file, entering HTML in, 491
- HelpViewer application
 - features of, 485
 - using *help:* protocol, 503–506
- Hide Recipe Info menu items, using, 313–316
- Horizontal Split View, resizing, 104
- HTML files, formatting, 34–35

I

- IB (Interface Builder) application. *See* Interface Builder
- Icon Composer, launching, 354
- icons
 - adding, 353–356
 - finding graphic images for, 143–144
 - size limit for, 353
- if test
 - placing in parentheses, 104
 - for restoring autosaved documents, 288, 292
- images. *See also* graphic images
 - availability of, 84
 - controlling scaling of, 145
- implementation files. *See also* method implementations
 - declaring class methods in, 217
 - main.m file in, 23
 - Objective-C methods in, 23
 - organizing, 18
 - revising, 21–23
- #import preprocessor directive, changing, 23
- Importer folder, displaying in Finder project window, 16
- indenting wrapped lines, 19–20
- index, fixing, 48
- Indexer utility, using with help book, 494
- indexing, controlling in Xcode, 19
- Info.plist file
 - adding diary document to, 115–121
 - declaring UTI (Uniform Type Identifier) for, 118–119
 - format of, 36
 - incrementing CFBundleVersion value in, 91
 - opening, 36
 - opening contextual menu for, 36–37
 - required settings, 41–42
 - uses of, 36
 - using generic name for, 35
- InfoPlist.strings file
 - editing, 42–43
 - localization contractor, 43–44
- inheritance, implementation via protocols, 170

- init method, using with DiaryWindowController, 101–103
 - initial first responder, explained, 146
 - initialization method, using with objects, 101–103
 - instance variable
 - declaring getter method for, 127
 - setting up for text view, 134
 - using in text systems, 124–125
 - Interface Builder
 - actions and outlets, 55
 - Button Size inspector, 141–142
 - choosing templates in, 97
 - creating DiaryWindowController class, 97–98
 - dragging Tab View objects in, 439–440
 - versus GUI design utilities, 53–54
 - launching, 97
 - Library window, 97–98
 - nib files, 54–55
 - opening DiaryWindow nib file in, 104
 - opening MainMenu.xib in, 112
 - opening nib files in, 97
 - Option-drag, 142
 - outlets and actions, 55
 - Print panel accessory view, 370–374
 - resizing diary window in, 145–146
 - Rich Text checkbox, 107
 - Scroll View Attributes inspector, 106
 - selecting text views in, 107
 - setting up File’s Owner proxy, 97
 - Split View Attributes inspector, 105
 - Text View Attributes inspector, 107
 - Window Attributes inspector, 104–105
 - Window Identity inspector, 105
 - Window Size inspector, 105
 - Interface Builder 3.2
 - action methods, 85
 - adding tab view, 79–80
 - archiving projects, 87
 - attaching drawer to document window, 81
 - Attributes button, 58
 - autocompletion feature, 62
 - autosizing behavior for split views, 69–71
 - autosizing behavior for tab view, 80
 - browser view mode, 62
 - building and running application, 87
 - choosing classes, 64
 - Cocoa Simulator, 65–66
 - Connections Inspector, 59
 - creating controls, 84
 - creating document window for drawer, 80–81
 - default objects, 59
 - delegate methods as hooks, 73
 - delegate outlet, 59
 - design surface, 57
 - displaying class categories, 64
 - displaying help tag for selections, 68
 - displaying toolbar items, 84
 - displaying Toolbar object, 64
 - document window for main recipe, 57
 - documentation, 88
 - dragging and dropping Toolbar objects, 64–65
 - dragging views, 69
 - Drawer Attributes inspector, 82
 - Drawer Connections inspector, 82
 - Drawer Content View object, 82
 - Drawer Size inspector, 82
 - editing attributes of split view, 68
 - editing nib files reference to document, 61
 - enabling struts, 72
 - File’s Owner proxy, 59–60
 - fixing widths of split-view panes, 73–75
 - hooking up Drawer object, 82
 - Horizontal Split View object, 78
 - Identity inspector, 61
 - implementing actions, 84–85
 - Inspector window, 57–58
 - instantiating objects, 63–64
 - Layout views, 67
 - Library window, 64, 81
 - Library window palette, 57
 - listing objects, 64
 - looking at nib files, 56–59
 - My Document Connection inspector, 60
 - notifications as hooks, 73
 - NSDrawer class, 81
 - Objects tab view for drawer, 81
 - outline view mode, 62
 - owners of nib files, 60
 - positioning split view, 68–69
 - proxy for File’s Owner icon, 60
 - resizing split view, 69
 - resource for, 7
 - responding to changes in views, 73
 - saving projects, 87
 - selecting title bar, 59
 - selecting view objects, 68–69
 - setting autosizing behavior, 72
 - springs and struts for horizontal split view, 78
 - springs and struts for vertical split view, 70–71
 - testing configuration of split view, 72–73
 - testing vertical split view, 78
 - undoing split views, 68
 - Vertical Split View object, 67–68
 - View connections inspector for drawer, 83
 - View Size inspector, 72
 - Window Attributes inspector, 58
 - Window object for drawer, 83
 - @interface directive, editing, 22–23
 - internationalization. *See also* localization
 - of display name, 351–352
 - preparing localizable strings for, 255–256
- ## K
- key view loop, explained, 146
 - keyboard focus, determining for search field, 187
 - Keyboard pane, accessing, 156
 - keyboard shortcuts, checking, 314
 - keywords, adding to search experience, 506–509
 - KVC-compliant method, using with diary entries, 564–565
 - KVO (key-value observing), using with view controller, 375–381

L

labels

- adding to Recipes tab, 441
- using with accessory views, 372–373

Leopard. *See* Snow Leopard

Line Wrapping, turning on, 19

Linked Frameworks subgroup, contents of, 17

localizable form, specifying strings in, 45

localizable strings, preparing for internationalization, 255–256

Localizable.strings file, creating, 45–46

localization, using lproj folders for, 15–16. *See also* internationalization

localization contractor

- function in InfoPlist.strings file, 43
- turning over applications to, 45

localization files, editing, 16

localizations, accessing for help book files, 485

lproj folders

- using for localization, 15–16
- using with localizable strings, 45

M

.m file extension, using with implementation files, 23

MacHelp.help bundle, location of, 487–489

macros, using with localizable strings, 45–46

main menu. *See also* Recipe Info menu item

- application delegate class, 194
- archiving project, 213
- getting string with path for RTF file, 197
- overview of, 193–194
- Read Me item for Help menu, 197–198, 200
- saving project, 213
- targeting application package, 197
- testing, 197
- VRApplicationController class, 194–195
- writing action method, 196–197

MainMenu nib file, opening, 109

MainMenu.xib, opening in Interface Builder, 112

make command

- implementing for diary entry class, 566–568, 572–573
- using with objects in AppleScript elements, 564–565

master-master-detail view, 67

menu bar

- adding Character Viewer to, 156
- adding Diary menu to, 200–201
- opening mockup of, 109, 115

menu divider, creating, 220

Menu Item object, dragging to Window menu, 208

menu items

- adding help tags to, 336–337
- alternating, 316–321
- enabling and disabling, 200, 223
- fixing validation problem with, 299–300
- testing, 304–305
- validating, 223, 319

method implementations, editing, 26–29. *See also* implementation files

methods. *See also* accessor methods; action methods

- adding to classes, 216–218
- autocompleting, 19
- examining header files for, 74
- getting help with, 74
- implementation of, 152
- relationship to protocols, 169
- using +VR_ prefix with, 263

model

- in MVC design pattern, 121–122
- separating from view in MVC design pattern, 276

model- versus view-controller, 126

model-controller

- in accessor method, 126
- NSDocument class as, 91

Models group, expanding in Xcode project window, 15

MVC design pattern. *See also* design patterns

- considering for Add Tag push button, 161
- considering for -addEntry: action method, 149
- controller in, 126
- function in document-based applications, 21
- function of window controller in, 31
- model in, 121–122
- model-controller, 126
- separating model from view, 276
- view-controller, 126

N

\n escape sequence, including in format string, 156

namespace collision issue, dealing with, 263

navigation buttons

- connecting action methods, 178
- declaring and implementing range methods, 178
- disabling, 179
- setting class to ValidatedDiaryButton, 179
- testing, 180
- writing action methods, 178

New menu item, creating for Chef's Diary, 114–115

next responders

- saving for Recipe Info menu item, 211
- searching chain of, 199

nib files

- adding to Xcode projects, 99
- editing references to documents, 61
- examining in Interface Builder 3.2, 56
- fixing for applications enabled in Leopard, 361–362
- formats, 100
- in IB (Interface Builder) application, 54–55
- localizing, 46
- managing for DiaryWindowController class, 99
- managing in diary document, 94
- opening in Interface Builder, 97
- owners in Interface Builder, 60
- using with Recipe Info menu item, 209

nil result

- returning for date picker, 182–183, 185
- returning for main menu, 197

- returning for search field, 189
- testing for in `DiaryWindowController`, 103–104
- notification center, using with MVC model, 276–278
- notification method, implementing for autosaved documents, 294
- notification object, using with delegate methods, 135
- notification strings, declaring for autosaved documents, 293–294
- notifications. *See also* delegate methods
 - versus delegation, 277
 - ensuring sending for methods, 152
 - limitation of, 277
 - posting, 277
 - posting for autosave operations, 288–289
 - responding to, 278
 - using as hooks in Interface Builder, 73
 - using blocks for, 330–334
 - using with General tab view item, 453–455
- NULL, passing to error parameters, 253

O

- object controller, binding for, 605
- object-first commands, encrypt and decrypt, 578–580
- Objective-C language
 - basis of, 4–5
 - blocks API, 321–323
 - categories, 218
 - described, 3–4
 - method call, 86
 - procedure, 86
 - receiver, 86
 - runtime, 4
 - use of protocols in, 169–170
 - version 2.0, 5
- Objective-C methods, using in implementation files, 23
- object-oriented programming, resource for, 4
- objects. *See also* data-bearing objects; first responder object
 - designating as temporary delegates, 159
 - initializing, 101–103
- observers, registering and unregistering, 332–334
- outlets and actions, using with Find submenu, 204–205. *See also* Target-Action design pattern

P

- page border, printing, 421
- Page Setup menu item, removing, 396
- Page Setup panel, features of, 367
- pane splitter problem, fixing, 362
- panels, adding help buttons to, 509–510
- parentheses (`()`), using with `if` statement, 104
- PDF files, generating, 310
- pedantic flag, setting, 104
- placeholders, using, 149, 155–156
- PNG-24 format, using with icon images, 354
- PowerPC hardware, enabling applications on, 357–359

- `#pragma mark` statement
 - versus comments, 216, 221
 - creating for error handling, 243
 - using in current diary document, 224
 - using in document behavior, 219–221
 - using with autosaved documents, 294
- preference setting, displaying, 476
- preferences
 - adding to Chef's Diary document, 442–443
 - best practices for, 438–439
 - formatter settings, 441
 - modeless, 438
 - using segmented controls, 439
 - using tab views, 439–440
- preferences window
 - centering horizontally, 447
 - configuring first responder, 444
 - controls in, 440
 - separating contents of, 439
 - setting title for, 448
 - testing Recipes pane of, 469
 - updating for changes in Print panel, 473
 - user interface elements in, 444
 - using global variables in, 451–452
- preferences window controller, creating in Xcode, 445–449
- print info object
 - getting from current print operation, 411
 - modifying, 393
 - using with accessory view controller, 377–378
- print operations
 - getting current versions of, 410
 - saving, 394
- Print option, error associated with, 368–369
- Print panel
 - accessory view, 370–374
 - adding accessory view controller to, 381–389
 - adding help button to, 509–510
 - closing, 393–394
 - closing for accessory view controller, 385
 - with custom settings, 434
 - with default settings, 432
 - features of, 367
 - features pop-up menu, 368
 - with last page of document, 433
 - organizing principle, 367–368
 - Presets pop-up menu, 435
 - scaled to 75%, 434
 - set to landscape orientation, 433
 - updating preferences, 473
 - users' changes in, 473
- print scaling
 - alert for limit of, 458
 - application-modal alert, 426
 - examining behavior of, 422–423
 - implementing, 424–431
 - reasons for, 421–422
 - repositioning headers and footers, 428–430
 - requirements for, 423
 - suppressing corner marks, 427–428

- print settings
 - declaring `+initialize` method, 392
 - registering default values of, 391
 - saving customizations, 389–397
- print view
 - basis of, 401–402
 - creating `DiaryPrintView` class, 399–401
 - flipping, 413–414
 - instantiating and initializing, 399
- print view's frame, setting, 411
- printable contents, paginating, 406–409
- printing
 - current entry, 403–404
 - current selection, 404
 - custom headers and footers, 415–421
 - customizations to Chef's Diary, 369–370
 - diary document, 369
 - document content, 403
 - multipage documents, 398, 409
 - page border, 421
 - turning off centering in, 412
- printing settings, displaying user defaults, 471–472
- project state, keeping track of, 94–96
- project windows, comparing, 15
- projects. *See* Xcode projects
- properties
 - declaring, 599–603
 - `readonly`, 602
 - using, 601
- protocol list
 - use of angle brackets in, 176
 - using in user interface validation, 174
- Protocol Reference document, consulting, 137
- protocols
 - conforming in user interface validation, 174
 - declaring for Add Tag push button, 177
 - declaring via categories, 218
 - testing conformity to, 176
 - use in Objective-C language, 169–170
- PSD format, saving icon images in, 353
- Push Buttons, dragging to diary window, 140–141

R

- radio groups, using with accessory views, 372–373
- Read Me file, opening, 212
- Read Me item, adding to Help menu, 197–198, 200
- recipe document, autosaving toolbar configuration for, 284–285
- Recipe Info button, placing in toolbar, 85
- Recipe Info command, using, 213
- Recipe Info menu item. *See also* main menu
 - adding drawer to responder chain, 210
 - building and running application, 212–213
 - connecting to action, 208
 - opening Window menu for, 208
 - resolving responder chain issue, 208–209
 - setting target and action, 209
 - using nib files, 209
 - validation of, 212
 - writing action method, 209–210

- Recipe Info toolbar item, features of, 84–85
- Recipe Markup Language website, 612
- Recipe window drawer. *See also* drawer
 - adding to responder chain, 210
 - containing window for, 266
 - determining screen space for, 265
 - getting maximum width of, 266
 - `NSDrawer` object, 265
 - opening, 208–213
- Recipe Window object, Window Connections inspector, 83
- recipes applications, availability of, 611–612
- recipes documents, creating, 109
- Recipes pane of preferences window, testing, 469
- Recipes tab, adding label to, 441
- Recipes tab view item
 - features of, 459–460
 - Use Current Size button, 465–467, 469
 - writing outlets for, 460
- recipes window. *See also* diary window; document windows
 - changing standard state of, 463–465
 - displaying standard state of, 460–462
 - displaying width and height of, 462–463
 - getting size of, 460–462
 - setting initial size of, 269
 - setting maximum size of, 262
 - setting minimum size of, 261
 - toolbar in, 267
- Redo action, testing for Add Entry push button, 157
- Redo menu item, using with `-addEntry:` action method, 152
- release notes, reading, 138
- Rename option, using with Xcode projects, 24
- renamed files, locating in Xcode, 25. *See also* files
- represented object design pattern, 377–378
- Resources group, displaying files in, 16
- responder chain
 - adding Recipe window drawer to, 210
 - altering for Recipe Info menu item, 211
 - implementing in Diary Tag Search menu item, 205–206
- responder chain, overview of, 198–200
- reverse DNS name, using in Info.plist file, 39
- Revert to Saved menu item
 - action method triggered by, 300
 - choosing, 303
 - implementing, 298–304
 - testing, 303–304
- Rich Text checkbox, using, 107
- RTF files
 - convenience of, 117
 - formatting, 33–34
 - using with main menu, 195, 197
- RTF formatting capabilities, applying, 107
- RTF text, storing and retrieving, 130
- RTF text view, displaying, 120

S

- sales transactions, managing, 594
 - Save As PDF menu item
 - accessing, 307
 - adding print settings dictionary, 312
 - considering, 308
 - displaying in File menu, 310
 - setting default name for, 347–349
 - save operations, best practices for, 157–158
 - Save panel, providing default name in, 345–346
 - saved documents, reopening, 274. *See also* documents
 - scaling images, controlling, 145
 - Scroll View Attributes inspector, using with DiaryWindow nib file, 106
 - scrolling text views, adding to diary window, 104–108. *See also* text view
 - sdef files
 - advisory about editing of, 525–526
 - consulting documentation for, 525
 - creating, 522
 - search experience, adding abstracts and keywords to, 506–509
 - search fields
 - adding placeholder text to, 187
 - array of tag ranges, 188
 - connecting, 190
 - controlling instances of same tag, 188
 - determining keyboard focus, 187
 - dragging into diary window, 142
 - running and testing, 190
 - validating, 190
 - Search menu items, conventions for, 202
 - search tag, creating, 188–189
 - selectors, using in user interface validation, 174
 - Set Tag button
 - changing class to ValidatedDiaryButton, 177
 - conforming to VRValidatedControl protocol, 176
 - setter accessor methods, writing, 600
 - setter method, using in text system, 125–126
 - Show Recipe Info menu items, using, 313–316
 - signatures
 - pattern for action methods, 148
 - use of, 41
 - singleton model, using with accessory view controller, 376
 - Snapshot facility, using with Chef's Diary, 94–96
 - Snow Leopard
 - enabling applications in, 357–364
 - features of, 2–3
 - fixing pane splitter problem, 362
 - preventing duplicate code, 76–77
 - reusing help files, 512
 - revising help files, 512–513
 - verifying for current diary document, 227, 229
 - software, distributing, 592
 - sort order enumeration, using with verb-first commands, 575–576
 - SortCommand class, creating, 577–578
 - split view
 - configuring for diary window, 133–135
 - creating outlet for, 283
 - exposing lower panes of, 121
 - managing in empty window, 282–284
 - saving divider position for, 284
 - setting up autosizing behavior for, 107–108
 - testing panes of, 134
 - views in, 127
 - Split View Attributes inspector
 - opening, 362
 - using with DiaryWindow nib file, 105
 - SQLite document types
 - expanding, 37
 - using with Core Data applications, 37
 - Standard Suite
 - make command, 564–565
 - reading, 524–525
 - verb-first commands, 574–578
 - view of application class, 535
 - Stepper Attributes inspector, using, 442
 - strict singleton, using with accessory view controller, 376
 - string conversion, using with zoom size for windows, 272–273
 - string files, use in localization, 255
 - strings
 - defining, 156
 - specifying in localizable form, 45
 - using as keys, 120
 - using macros with, 45–46
 - struts, enabling in Interface Builder, 72
 - subclasses, using with documents, 20
 - subfolders, creating in Xcode, 14
 - subgroups, creating in Classes group, 92–93
 - sudden termination, adding support for, 350–351
 - suppression checkboxes, setting up, 449
 - system font, obtaining, 417
- ## T
- tab order, controlling, 146–147
 - tab view item
 - displaying user default printing settings, 471–472
 - implementing Autosaving section, 474–477
 - tab views, using, 439–440
 - Tag All button, using dynamic Add tag with, 323–330
 - Tag All menu item, using, 316–321
 - tag list, marking for Add Tag push button, 160
 - tag titles, obtaining range for Add Tag push button, 162
 - tags
 - controlling instances in search field, 188
 - function of, 186
 - searching in text, 188
 - separating, 160
 - Target-Action design pattern, overview of, 86. *See also* outlets and actions
 - Targets group, contents of, 17–18

- templates
 - choosing in Interface Builder, 97
 - features of, 20
 - naming for document-based applications, 22
 - selecting for window controller, 31
 - using in Xcode, 12
- termination, adding support for, 350–351
- terminology dictionary
 - adding HTML documentation to, 534, 536
 - creating, 520–525
 - reading during development, 524–525
- terminology suites
 - availability of, 525
 - Text Suite, 542–544
- text
 - reading from disk, 130–132
 - searching tags in, 188
 - writing to disk, 130–132
- text storage object, accessing, 127
- Text Suite
 - adding, 542–544
 - features of, 542–543
- text system
 - allocating memory for NSTextStorage object, 132
 - design of, 126
 - MVC in, 122–123
 - storing document attributes, 132
 - using accessor methods in, 124–125
 - using getter method in, 125–126
 - using instance variables in, 124–125
 - using setter method in, 125–126
- text view. *See also* scrolling text views
 - accessing, 127
 - best practices for, 151
 - connecting outlet to, 127
 - constructing, 123
 - selecting in Interface Builder, 107
 - setting up instance variable for, 134
 - verifying selection of, 146
- Text View Attributes inspector, using with
 - DiaryWindow nib file, 107
- TextEdit
 - opening Read Me file in, 212
 - problems with sample code, 157–158
 - using, 133
- time styles, setting up, 155
- toolbar
 - customizing, 84
 - placing Recipe Info button in, 85
- toolbar configuration, autosaving, 284–285
- toolbar items
 - adding help tags to, 337
 - displaying in Interface Builder, 84
- Toolbar object
 - displaying in Interface Builder 3.2, 64
 - dragging and dropping in Interface Builder 3.2, 64–65
- tooltips, adding, 334–337
- topic page, adding AppleScript link to, 502–503
- Trash, verifying files in, 236

U

- Undo action, testing for Add Entry push button, 157
- undo coalescing, breaking, 157
- Undo menu item, using with `-addEntry:` action method, 152
- Unicode characters, finding, 156
- URL. *See* file URL
- user defaults, saving structures to, 272
- user interface
 - copying elements of, 142
 - master-master-detail view, 67
- user interface validation
 - conforming protocols, 174
 - performing, 168–169, 172–173
 - types of, 171 protocol; validation
 - using protocol list, 174
 - using selectors in, 174
- user support, providing, 590–591
- `userInfo` dictionary
 - declaring keys for, 248
 - getting error from, 246
 - using key-value pairs in, 250
- UTI (Uniform Type Identifier)
 - declaring for Info.plist file, 118–119
 - legal characteristics of, 117
 - looking up, 116

V

- validation
 - function of, 171–172
 - of menu items, 299–300
- verb-first command, sort, 574–578
- Vermont Recipes 2.0.0* file, creating, 22
- Vermont Recipes 2.0.0* folder, opening for IB 3.2, 56
- Vermont Recipes* application. *See also* applications
 - accessing folder for, 14
 - archiving, 51, 191
 - Build pane of, 48–49
 - building, 51
 - building for release, 586–587
 - changing build settings, 49
 - Configuration pop-up menu, 49
 - creating folder for, 14
 - Debug settings, 49
 - displaying settings, 49
 - distributing, 591–594
 - expanding Target group for, 50
 - General pane of, 47
 - overview of, 9–10
 - promoting, 595–596
 - providing documentation, 589–590
 - providing user support, 590–591
 - Release settings, 49
 - saving project for, 13
 - selecting, 46
 - testing, 587–589
- Vermont Recipes* GUI. *See* Interface Builder 3.2

- Vermont Recipes Suite
 - adding suite definition for, 526–527
 - implementing AppleScript terminology, 531
 - implementing **terminology version** property, 529–531, 534
 - naming getter for, 530
 - problem with SLOG messages, 533–534
 - testing **terminology version** property, 532
 - warning, 527
- versions, indicating, 39–40, 77
- view MVC element, function of, 30
- view- versus model-controller, 126
- view-controller, DiaryWindowController, 91
- views
 - controlling selection of, 146–147
 - resources for, 63
- VoiceOver utility. *See also* accessibility features
 - auditing performance in, 345
 - Verbosity pane of, 340
- +VR_ prefix, using with methods, 262–264
- .vrdiary files, choosing, 133

W

- Window Attributes inspector, using with
 - DiaryWindow nib file, 104–105
- Window Connections inspector, using with Recipe
 - Window object, 83
- window controller. *See also* document windows
 - creating and revising files, 31–32
 - delegate methods, 30–31
 - features of, 30–31
 - initializing for DiaryWindowController class, 100–104
- Window Controllers subgroup, creating for Chef's
 - Diary, 92
- window frames
 - autosaving, 275, 280–282
 - setting autosave name for, 288
- Window Identity inspector
 - Notes field, 105
 - using with DiaryWindow nib file, 105
- Window menu
 - dragging Menu Item object to, 208
 - opening for Recipe Info command, 208
- Window object, using with drawer, 83
- Window Size inspector
 - using, 260–261
 - using with DiaryWindow nib file, 105
- window size, saving, 271, 273
- windows. *See also* diary window; document windows
 - capturing standard state from nib file, 271–274
 - saving standard size of, 272
 - setting standard state of, 271–274
 - standard state of, 459
 - user versus standard states of, 269
- workspace, configuring, 18
- wrapping lines in Xcode, 19

X

- Xcode
 - archiving projects, 51
 - build settings, 47
 - building and running application, 51
 - Building pane, 19
 - choosing application type in, 13
 - Code Sense pane, 19
 - controlling indexing, 19
 - creating accessory view controller in, 374–381
 - creating DiaryDocument class in, 91–94
 - creating preferences window controller in, 445–449
 - creating projects in, 12
 - creating view controller in, 374–381
 - creating VRDocumentController class files, 111–112
 - documentation, 52
 - Documentation pane, 20
 - features of, 11
 - Groups & Files pane, 14
 - Indentation pane, 19–20
 - launching to add controls, 140
 - location of project files in, 15
 - organizing principle, 15
 - pointing to renamed files, 25
 - preferences, 47
 - release notes for 3.x, 52
 - resource for, 7
 - saving projects, 51
 - searching for declarations in, 48
 - setting options for applications, 13
 - setting preferences, 18–20
 - setting up New File window, 31
 - Snapshot facility, 94–96
 - templates, 12
- Xcode project window
 - Classes group, 17
 - expanding Models group in, 15
 - features of, 14
 - Groups & Files pane, 18
 - Importers subgroup in, 17
 - Linked Frameworks subgroup, 17
 - nesting groups in, 17
 - Other Frameworks subgroup, 17
 - Other Sources group, 17
 - renaming files in, 25
 - Targets group, 17–18
- Xcode projects
 - adding nib files to, 99
 - changing filenames in, 24–25
 - choosing Rename option, 24
 - creating, 12–13
 - examining changes to, 94–96
 - keeping in folders, 13–14
- .xib file extension, explained, 54

Z

- zooming windows, 269