Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Introduction

NEXT >

Vermont Recipes is a cookbook for developing Macintosh computer applications in the Mac OS X Cocoa environment using the Objective-C programming language. It takes a practical, no-nonsense, hands-on, step-by-step approach, walking you through the details of building a Cocoa application from start to finish. It explains in detail what the code is doing and why it works, but it offers a minimum of theory about the language or the Cocoa frameworks—for that, you are encouraged to read Apple's Mac OS X Developer Documentation. *Vermont Recipes* places a decided emphasis on getting an application to work correctly as quickly as possible.

The current version of *Vermont Recipes* is written for Mac OS X 10.0, which was released on March 24, 2001 and has been updated with minor bug fixes and feature enhancements several times since then. To make use of the Cookbook to create an application, you must install the Mac OS X Developer Tools, which have also been updated since March 24.

If you studied the previous version of *Vermont Recipes*, which was written for Mac OS X Public Beta, there is little to gain by going through the first three Recipes again. There have been very few changes to the code of the Vermont Recipes application, which are listed on the Errata and Updates page. Although this version of *Vermont Recipes* does walk the reader through many of the improvements that have been made to Apple's Developer Tools in Mac OS X 10.0, you can learn about these changes by starting with *Recipe 4* and following as they become available.

Vermont Recipes is aimed at reasonably experienced programmers who are Cocoa beginners. It assumes a solid grounding in the C programming language and some experience with object-oriented programming concepts. No prior experience with Cocoa's predecessors, NextStep, OpenStep, and Rhapsody, is

necessary. Only a limited exposure to the Objective-C extensions to C is required, along with a modest awareness of the workings of Apple's Mac OS X development tools, Interface Builder and Project Builder. If you already know C and have some understanding of object-oriented programming, the only preparation you should undertake is a little reading: *Inside Mac OS X: System Overview* for essential background information on the architecture of the Mac OS X operating system; *Inside Mac OS X: Object-Oriented Programming and the Objective-C Language*, Apple's Objective-C documentation; and Apple's developer tools documentation relating to Interface Builder and Project Builder. All of these materials are available on your computer once you install the Developer Tools CD. You will then be ready to start cooking with *Vermont Recipes*.

How To navigate *Vermont Recipes*

Every page in *Vermont Recipes* contains navigation bars near the top and bottom. On the left end of a navigation bar are links taking you back up the site hierarchy from the page you are viewing. Using these links, you can always return immediately to the first page of the Recipe you are reading, the detailed Table of Contents, and this Introduction. On the right end of a navigation bar are links taking you "BACK" to the previous page and forward to the "NEXT" page in sequence.

How to read *Vermont Recipes* off line

If you prefer to work off line, you can download *Vermont Recipes* as a Portable Document Format (PDF) file and read it using the free Adobe Acrobat Reader. This PDF file contains the entire Cookbook and is rather large. You can download a smaller PDF file containing an individual Recipe at the first page of each Recipe.

Return to the Stepwise site at *www.stepwise.com/Articles/VermontRecipes/* from time to time to check for new and updated *Vermont Recipes* pages. Every page's last modification date appears near the top. The PDF files are synchronized with the web site within a few days of every change.

Errata

From time to time, errors will be discovered in *Vermont Recipes*. Errors are fixed promptly when discovered, and the relevant text and code is revised. To assist those who relied on earlier versions, an Errata and Updates page is provided which lists all substantive code errors, their corrections, and the date of the correction. If you consult the Errata and Updates page online occasionally, you will be able to keep your code up to date with the latest fixes. (Corrections to typographical errors in text are not listed.)

The emphasis here is on code. *Vermont Recipes* is not a tutorial on how to use Interface Builder or Project Builder, although the steps necessary to create an application are detailed in *Step 2* of the first Recipe with enough particularity to get you started. Nor does it explain Objective-C syntax, except for an occasional brief discussion of important features of the language in the Cocoa context, such as categories and protocols. Instead, it is a collection of simple, do-it-yourself Recipes—not much more than commented and organized code snippets—to guide you through the process of creating classes and subclasses, objects, outlets, and actions.

Each Recipe is accompanied by downloadable project source files so you can follow along if you prefer not to type the code yourself. Declarations and definitions in the source files are annotated with references to the Recipes and Steps where they are described; if you are a nonlinear thinker, you can start with the source files and look up the explanations in the Cookbook.

There is no "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 be easily adapted to a variety of scenarios. It attempts to conform as much as possible to conventional practices and nomenclature. The application you will build here is a so-called document-based application using Cocoa's AppKit and Foundation frameworks, which gives it the flexibility and capability to serve as a model for the widest variety of applications.

Most importantly, *Vermont Recipes* was begun while I was learning Cocoa through careful study of Apple's 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 otherwise be confusing and frustrating to a Cocoa beginner. It also benefits from the years of experience of Scott Anguish and the Stepwise team, who have graciously reviewed and commented on the code and this text. My thanks go to Scott and the team for helping to make this tutorial conform as closely as possible to established Cocoa techniques. Any errors are mine alone.

Why Cocoa?

Apple is understandably emphasizing Carbon development as Mac OS X is being rolled out, in order to

encourage rapid migration of existing applications from Mac OS 9 to Mac OS X. The Carbon application environment allows those having 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.

But Cocoa has been positioned as what might be called the "real" Mac OS X for the future. Apple has repeatedly urged developers of new Mac OS X applications to develop them using the Cocoa frameworks, and Apple's recommendation has grown stronger with the rollout of Mac OS X 10.0. These are mature and powerful application frameworks based on ten years or more of NextStep and OpenStep experience. They incorporate, or soon will incorporate, all of the new functionality and interface of Mac OS X. Furthermore, with Cocoa you can have it both ways: all of the Carbon APIs can be called from within a Cocoa application. Finally, experienced developers report that the Cocoa frameworks reduce development time by a very substantial factor. In short, it is faster and easier to develop new applications in Cocoa, with no loss of power or flexibility.

Why Objective-C?

You can use Java 2 for Cocoa development, but many Cocoa developers prefer Objective-C. Java is not, any more than Objective-C, an answer for those who need to develop desktop applications in a cross-platform environment, because the Cocoa frameworks are not cross-platform. Furthermore, while Apple is working hard to improve its performance, knowledgeable developers continue to express concern about Java's speed in a desktop environment. Apple is nevertheless doing a good job of implementing almost all of Objective-C's unique features in Java, and it has marketed Cocoa as a powerful Java platform. Java may be an attractive language for Cocoa development, especially for those who already know it, and especially for those doing web application development with WebObjects 5, which is based on Java.

Vermont Recipes is based solely on Objective-C. Objective-C is a surprisingly easy language to learn if you already know C. A widely-repeated rule of thumb is that a C programmer can learn the Objective-C extensions in a day or two. Choosing to develop Cocoa applications in Objective-C may therefore be motivated by nothing more than the wish to avoid the substantial investment of time required to learn a fundamentally new language like Java. There are much more substantial reasons to use Objective-C, however. In particular, 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 languages.

Naming conventions

Vermont Recipes attempts to follow the naming conventions of Objective-C as they have grown up around NextStep and its successors. Some of these conventions are actually required in order to take advantage of built-in features of the Cocoa development environment. Others are work habits that have become more or less generally accepted in the Objective-C and Cocoa communities because they make it easier to read other developers' code. The following are only the most common rules, collected from miscellaneous postings to the Cocoa mailing lists. They appear in no particular order.

- Give an accessor method the same name as the variable it accesses. For example, method `myName` to get variable `myName`.
- Give a method that sets a variable a name beginning with "set" followed by the name of the variable with an initial capital letter. For example, method `setMyName:` to set variable `myName`.
- Start method names with a lowercase letter. For example, `init`.
- Start class names with an uppercase letter. For example, `MyDocument`.
- Prefix exported variables, notification names and the like with two or three distinctive letters defined for the bundle, to avoid contaminating the global name space. For example, `VR` for Vermont Recipes, `NS` in most of the Apple Cocoa classes, `EO` for Enterprise Objects classes, and `WO` for Web Objects classes.

Apple reserves to itself the use of a leading underscore character when naming private methods and exported functions. If developers were also to use this naming convention, they would risk unknowingly overriding private methods in Apple's frameworks.

A note about Interface Builder and Project Builder

A Cocoa beginner may perceive Interface Builder to be nothing more than an interactive graphical user interface design tool and Project Builder to be the real tool for building an application. This perception would be inaccurate, but in *Vermont Recipes* you will nevertheless be taught to start the application development process by using Project Builder to create a new project and to write its code. Interface Builder will be used here mainly as a tool to design and build the graphical user interface, but not to generate significant amounts of code. Interface Builder's Read Files command will be used periodically to update the internals of the Interface Builder nib files whenever outlets and actions have been added, deleted or modified in the source code. But Interface Builder's Create Files command will be used rarely, and then generally only for prototyping a new class.

Eventually, greater integration of these tools will allow you to use Interface Builder more heavily, not just to design and build the user interface, but also to generate and update the code for classes, outlets, actions and other items, automatically. For now, in the current version, you are cautioned not to use the Create Files command when your project already contains source code, because it may overwrite your existing files.

You will nevertheless discover that the nib files generated by Interface Builder are an integral and necessary part of a Cocoa application. A nib file describes an application's user interface more comprehensively than would a simple design tool. Interface Builder allows you, for example, to use intuitive graphical techniques to tell your code which user controls are connected to specific instance variables, or "outlets," in your code, and which methods, or "actions," in your code are triggered by specific user controls. A nib file is more than a collection of generated code to be compiled with your application code; it is, in fact, an archived set of classes, instantiated objects and connections that a Cocoa application loads 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

can use Interface Builder, for example, to alter the user interface of a compiled application and, conversely, to prototype and test a user interface without compiling an application.

The *Vermont Recipes* application specification

Because the target audience of *Vermont Recipes* is a diverse group of programmers who plan to pursue a wide variety of projects, the subject of the Cookbook is a generic application implementing all of the features that are typically found in many applications and utilities. These include multiple documents and windows, all manner of user controls, menus, tabbed views, and drawers, and standard Macintosh techniques such as drag-and-drop editing. It will not be a focused, topical application designed to serve any particular purpose, such as a music notation tool or a checkbook balancing application. Instead, it will serve simply as a showcase for common user interface devices, demonstrating not only how to build them but also how they work when completed. In this way, it is hoped that programmers will find information here that they can profitably use in their own applications, even though the Vermont Recipes application itself won't actually "do" anything at all.

The application is therefore specified, in broad strokes, as follows: It allows multiple documents to be open simultaneously. Each document can be saved separately with its own settings using a common file format. Each document is represented by a standard main window containing several tabbed views showcasing various categories of user controls, with additional slide-out drawers for ancillary controls. Each document also allows additional windows to be opened, to provide, for example, a large, scrollable space for typing text. As time goes by, the application will acquire additional features, such as a full-blown Help system, and it will become scriptable and "speakable." In short, if you're planning to create a simple, multi-document, multi-window application, the Vermont Recipes application will provide a usable model.

You, too, can tell the world your product runs on Mac OS X! The art work, licensing requirements, and guidelines for use of the "Built for Mac OS X" badge are available on the ADC Software Licensing web site. Please note that this badge cannot be used for products that launch the Classic environment.

Installing the downloadable project files

Each Recipe includes a link to the full project files up to that point in downloadable form, in case you do not want to type the code as you read through the cookbook. To use the project files, you must have installed the Mac OS X Developer Tools CD, which is included with the Mac OS X product when purchased as a separate product.

The download is in the form of a standard Macintosh single-fork disk image file. After downloading it, you will find a file in your download folder or on the desktop named, for example, `VermontRecipes1.dmg`. If it does not mount automatically as a disk in Mac OS X and open, just double-click the image file to mount it and double-click the mounted disk icon to open it. Then drag the `Vermont Recipes` folder that it contains to the place where you keep your development files (for example, in your home `Documents` folder).

NOTE: To mount the disk image, you may have to drag the image file to the Mac OS X version of Disk Copy if this is the first time you have used that utility. If you double-click the image file, instead, it may be opened by the Mac OS 9 version of Disk Copy and refuse to install. The Mac OS X version of Disk Copy is found in the `/Applications/Utilities` folder. Once the image is mounted, you will find it either on your desktop or by clicking the Computer button in a Finder window.

Information resources

After installing the Mac OS X Developer Tools CD, you will find a great deal of information about developing Cocoa applications in the `Developer` folder, at the root level of your Mac OS X disk, and in Apple Help. You should read almost all of it, including especially the Release Notes and the AppKit and Foundation frameworks documentation. In particular, you are strongly advised to read the documentation for the Cocoa `NSDocument` and `NSWindowController` classes before beginning.

Specific references will be given in a "Documentation" section in some of these pages, as may be appropriate to the topic at hand. Most refer to files installed with the Developer Tools CD. As time goes by, the files installed with the Developer Tools CD will become obsolete, so be sure to check the Apple Computer sites listed below for updates from time to time.

As you work through each of the specific Steps in these Recipes, it is very important that you read the relevant AppKit and Foundation developer documentation for the classes, protocols, methods, and functions used in each Step. These *Vermont Recipes* are not much more than step-by-step "how-to" instructions, with little explanation, because the developer documentation already provides thorough explanations. The easiest way to get into the developer documentation is to go to the Cocoa page in the Developer Help Center in the Apple Help Center; click on Application Kit or Foundation under "Objective-C Framework Reference" in the Reference Documentation section. The same material is organized more topically in the specific references listed at the bottom of the same page, which are also available in */Developer/Documentation/Cocoa/TasksAndConcepts/ProgrammingTopics* on your computer. Don't forget to check the Apple Computer sites occasionally for updates.

It can also be helpful to examine the AppKit and Foundation framework header files, which are in some cases commented to provide information not contained in the developer documentation. The header files are in */System/Library/Frameworks/AppKit.framework/Headers* and *Foundation.framework/Headers*.

Cocoa books are already beginning to appear. The first out of the gate was *Learning Cocoa* (O'Reilly, 2001) by Apple Computer, which is an updated and expanded version of earlier Next documentation and tutorials, extensively illustrated. This book is highly recommended for its introduction to Objective-C, its lessons on how to use Project Builder and Interface Builder, and its thorough explanation of the relatively simple example applications described in the book. At least three other books on Cocoa development are planned for publication during 2001, two of them written by former NextStep and OpenStep developers with extensive experience and likely to appeal to a more advanced audience.

Cocoa training for developers is available from Apple and from third parties. Apple offers a five-day Apple iServices Cocoa Development program. Aaron Hillegass, who worked at Apple Computer, Inc. and NeXT Software, Inc. as the senior trainer and curriculum developer, offers a multi-day training program through his Big Nerd Ranch in Asheville, NC.

Information about Cocoa development is proliferating on the web. These are a few sites of general interest:

Apple Computer sites

Mac OS X Development, Apple Computer, Inc.

> The starting point for Apple's Mac OS X developer offerings.

Mac OS X Developer Documentation, Apple Computer, Inc.

> A table of contents for Apple's Mac OS X developer documentation, including Cocoa. Many items found here are also on the Mac OS X Developer Tools CD, but newer versions will appear on the web site over time.

Developer Essentials, Apple Computer, Inc.

> Captures in one place, for those new to Mac OS X development, all of the basic links relating to software development on Mac OS X, including the latest volumes of Inside Mac OS X.

Cocoa Developer Documentation, Apple Computer, Inc.

> A table of contents for Apple's Cocoa documentation. The What's New page is updated only occasionally and is not currently a useful page for keeping up to date with the current state of Apple's Cocoa documentation.

Sample Code, Apple Computer, Inc.

The entry point for Apple's offerings of sample code, including small example applications. A number of Cocoa code examples have been posted; expect many more in the days, weeks and months to come. A button is provided to filter the sample code for those relating to Cocoa.

Inside Mac OS X: Aqua Human Interface Guidelines, Apple Computer, Inc.

The official guide to graphical user interface design conventions for Mac OS X, frequently updated in these revolutionary times. Your application is not truly a Mac OS X application if it does not conform to these Guidelines.

Third-party Cocoa developer sites

Stepwise, Scott Anguish

A highly technical and very sophisticated site maintained by a long-time NextStep developer, containing contributions from many others. Besides hosting *Vermont Recipes*, Stepwise is full of learned and useful articles about NextStep, OpenStep, Rhapsody, and Cocoa development, including Scott's well-known tutorial based on an earlier version of Cocoa, HTMLEditor.

A Cocoa FAQ has been started on Stepwise. Send submissions to *cocoa-faq@stepwise.com*.

The Omni Group

A Cocoa community-oriented site for sample code, mailing lists, and other developer resources, as well as a number of mature Cocoa frameworks available for free subject to license terms, from a long-time publisher of NextStep, OpenStep, Rhapsody and Cocoa applications.

Stone's Throws, Andrew Stone

NextStep, OpenStep, Rhapsody and Cocoa tutorials and sample code, from a long-time developer.

Cocoa Dev Central, Erik Barzeski and others

Tutorials, tips, news, links and other information for new Cocoa developers.

CocoaDev, Steven Frank

A collection of links to Cocoa developer resources, plus user-contributed sample code.

Cocoa tutorials

See also the Apple and third-party developer sites, above.

DISCOVERING OPENSTEP: A Developer Tutorial (Rhapsody Developer Release), Apple Computer, Inc.

Apple's excellent Rhapsody tutorial, still relevant, and still the best place to learn how to use Interface Builder and Project Builder if you haven't yet bought the new O'Reilly book mentioned above.

Programming With Cocoa, Mike Beam

A series of articles for beginners at the O'Reilly Mac DevCenter site.

Cocoa Dev Central Tutorials, Erik Barzeski and others

A collection of small tutorials on miscellaneous tasks.

Programming Mac OS X with Objective-C and Cocoa, Steven Frank

A Cocoa tutorial by a beginner, for beginners, currently based on Developer Preview 4.

Cocoa links

Native OS X Applications—Cocoa Frameworks, Jeff Biggus

Links to Cocoa and other frameworks.

Native OS X Applications—Cocoa Source Code, Jeff Biggus

Links to Cocoa applications offered with downloadable source code.

Cocoa Programming Examples and Source Code, MacTelligence

Cocoa-related links, including some not in this list.

Cocoa (and older) sample code

Not all of these sites contain examples that work under Mac OS X. See also the Apple and third-party developer sites, and be sure to check the Cocoa Links section, above, for additional sample code not listed here. A listing here is not a guarantee of good code; use at your own risk!

OSX Quickies, Chuck Bennett

> Currently holds his code to emulate the old NextStep sndplay program that lets you play sounds from the command line.

Examples, Mmalcolm Crawford

> Downloadable Cocoa code snippets by a well-known figure in the Cocoa community who is currently teaching Cocoa to Apple Computer employees in Cupertino.

Mike's Free Stuff, Mike Ferris

> Downloadable OpenStep, Rhapsody, and Cocoa examples from the head of Apple's developer tools group. It includes MOKit 2.6, a framework recently updated for Mac OS X 10.0 containing several useful classes that augment the Cocoa frameworks. Paraphrasing the author, it includes a regular expression object, an `NSFormatter` subclass that uses regular expressions to validate strings, and a set of objects for implementing text completion. It also serves as an example of how to construct and distribute a framework. TextExtras, based on MOKit, uses the input manager bundle loading mechanism in Cocoa to get itself loaded into all Cocoa applications; once loaded, it adds many cool features to the Cocoa text system.

CocoaSampleCode, contributed by many, collected by Steven Frank

> A collection of Cocoa sample code contributed by many. This site, run by Steven Frank, allows users to add contributions directly.

BigShow, Aaron Hillegass

> BigShow is an XML-based presentation tool with available source code.

GraphicKit, John Hörnkvist

> "A comprehensive framework for graphics under Cocoa. It fills a gap in the application kit by providing a highly optimized yet flexible base for applications that need to display structured graphics. The GraphicKit is a reimplementation of the display engine in MagnaCharta, and has been designed for Mac OS X." AppleScript is supported. Currently in beta. License required.

Mac OS X Rhapsody examples, Gerard Iglesias

Rhapsody example code.

Mac OS X Cocoa CoreAudio demo, James McCartney

A simple sinewave oscillator controlled by GUI sliders, showing how to get audio out of Mac OS X in as few lines as possible.

babe the blue osx..., Sebastian Mecklenburg

A simple `NSFormatter` subclass to check validity of characters as they are typed, and a macro. More is promised.

Programming Page, Andreas Monitzer

Downloadable Cocoa examples.

Mac OS X (Server) Apps, Mulle kybernetiK

Downloadable Cocoa applications with source code.

epicware, Eric Peyton

Downloadable Cocoa applications with source code.

Mac OS X - Programming Examples, Raphael Sebbe

Several Cocoa examples from an enthusiastic recent computer science graduate.

Cocoa examples by request, Frederic Stark

A few Cocoa examples, with an offer to write what you want.

Basic Cocoa Document Based Apps, Daniel Staudigel

Sample document-based application code examples, including an NSTableView example.

Cocoa Objective-C on Mac OS X, Sven Van Caekenberghe

Downloadable Cocoa examples.

MiscKit, Don Yacktman and others

Free objects and other reusable software, contributed by many. Don announced at the WWDC 2001 Stepwise Birds of a Feather session that this venerable framework is being revived for Mac OS X 10.0.

Objective-C

The Objective-C FAQ

Frequently asked questions covering Objective-C on all platforms, downloadable via ftp, updated monthly.

The Objective-C newsgroup

An active newsgroup for discussion of Objective-C issues on all platforms.

Objective-C: Documentation

Links and information about programming with Objective-C.

Optimizing Objective-C Code

A series of technical articles on how to take advantage of the innards of Objective-C to improve efficiency.

NeXTstep, OpenStep, and GNUstep

NeXTstep 3.3 Developer Documentation Manuals

Cocoa's father. Getting pretty old, but still interesting and relevant.

OpenStep Specification

Cocoa's uncle. From 1994, when the OpenStep API was made public under license.

GNUstep

Cocoa's brother. The home of an active OpenStep-derived development project similar to Cocoa.

About the author

Bill Cheeseman is a retired Boston lawyer of some notoriety (did you read or see "A Civil Action"?) now living in Quechee, Vermont. He first experienced the joy of computing when, in 1964, his Harvard roommate was programming the PDP-1 at the Cambridge Electron Accelerator in Fortran, and they played the original Space War daily for a year. He began writing programs himself in the mid-1970s, first on the HP-25 programmable calculator, followed by the HP-41C. As a member of the national HP-41C users group, he wrote the first compiler of undocumented HP-41C commands. He subsequently programmed extensively in AppleSoft Basic, UCSD Pascal, Modula-2 and 6502 assembler on the Apple][and Apple //e, Business Basic and UCSD Pascal on the Apple ///, and a wide variety of languages, including Basic, Pascal, Object Pascal, Modula-2, C and C++, on a long succession of Macintosh computers. He is well known in the AppleScript community as webmaster of The AppleScript Sourcebook. Having retired from the practice of law at the end of 1999, he is now beginning the new millennium with a second career, programming full-time in Objective-C in the Cocoa environment of Mac OS X.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/introduction.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Contents

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/contents.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 7, 2001 - 11:30 AM

Errata and Updates

This page lists all significant corrections and changes that have been made to the published code of the Vermont Recipes application. If you built an earlier version, you can look here to find changes to correct it and bring it up to date.

Vermont Recipes for Mac OS X 10.0

The Vermont Recipes application as described in Recipes 1-3 was updated in a few respects on June 20, 2001 to accommodate changes in Mac OS X 10.0 over Mac OS X Public Beta. The list below is in Recipe/Step/instruction order. In addition, the inevitable errors are being reported and fixed.

- Recipe 1, Step 4.2.1, instruction 3. and Step 4.2.2, instruction 3., June 20, 2001. The Public Beta version of the application used ASCII encoding when converting and restoring data to save and retrieve it from persistent storage, in the `convertForStorage` and `restoreFromStorage` methods in `MyDocument.m`. Mac OS X 10.0 uses UTF8 encoding for property lists. Vermont Recipe's data is saved as a property list, so we have changed to UTF8 encoding, passing `NSUTF8StringEncoding` as the `encoding` parameter to NSString's `dataUsingEncoding` and `initWithData:encoding:` methods.

- Recipe 1, Step 5.2, instruction 1., August 4, 2001. In the first code snippet, the two instructions to localization contractors were reversed, and the error existed in the downloadable project files as well. The appliation nevertheless functioned correctly. The `Localizable.strings` file was correct.

- Recipe 2, Step 4, instruction 2.f., August 7, 2001. In the implementation of the `recentRockAction:` method, the "Set Recent Hits" string erroneously had a capitalized "I" in

"Hits." The application functioned correctly, but localization might not have worked correctly.

- Recipe 2, Step 4, instruction 2.i., August 7, 2001. The `restoreFromDictionary:` method of `MySettings.m` erroneously used the `recentRockValueKey` to obtain the value. It should use the `rockValueKey`. The error caused the application to malfunction, restoring the user interface incorrectly when reading a file from disk (the Allow Rock checkbox would appear to have the same value as the Recent Hits checkbox, even if it had been saved with a different value).

- Recipe 2, Step 5, instruction 2.i. and Step 6, instruction 2.i., June 20, 2001. The Public Beta version of Vermont Recipes left it as an exercise for the reader to implement categories on NSString to save and retrieve custom `VRParty` and `VRState` enumeration constants as strings instead of integers to improve the human readability of the documents. In the current version, we still leave it as an exercise for the reader, but you can now find the actual code in the `NSString+VRStringUtilities` files and calls to the new methods in the Persistent storage section of `MySettings.m`.

Vermont Recipes for Mac OS X Public Beta

A number of errors in the code of the original Mac OS X Public Beta version of the Vermont Recipes application were published, as noted below. The list below is in Recipe/Step/instruction order.

- Recipe 1, Step 3.5.1, instruction 6. December 31, 2000. The earlier instruction to place the `MySettings` header and source files in the `Resources` folder was erroneous. The `Classes` folder was intended. *Vermont Recipes* nevertheless functioned correctly.

- Recipe 1, Step 4.2.3, instruction 2. December 31, 2000. In earlier versions of `MySettings.m`, the `convertToDictionary:` method incorrectly read the `myCheckboxValue` variable directly instead of invoking its accessor method. *Vermont Recipes* nevertheless functioned correctly. The corrected code is as follows:

```
- (void)convertToDictionary:(NSMutableDictionary
*)dictionary {
  [dictionary setObject:[NSString
stringWithFormat:@"%d", [self myCheckboxValue]]
forKey:myCheckboxValueKey];
}
```

- Recipe 1, Step 5.1, instructions 3. and 4. December 31, 2000. In earlier versions of `MyWindowController.h`, the `windowWillReturnUndoManager:` delegate method incorrectly omitted its `sender` parameter. The corrected code is as follows:

```
- (NSUndoManager
*)windowWillReturnUndoManager:(NSWindow *)sender;
```

The method is also corrected in `MyWindowController.m`. *Vermont Recipes* nevertheless functioned correctly, although the delegate method was not invoked.

- Recipe 1, Step 5.2, instruction 2. December 31, 2000. In earlier versions of the `Localizable.strings` file, the comments to the Undo and Redo menu item names for the `myCheckbox` control did not match the comments in the invocations of the `NSLocalizedString:` method in `myCheckboxAction:` in `MyWindowController.m`, and the comments were reversed in the source file. *Vermont Recipes* nevertheless functioned correctly. You can correct this minor issue by referring to the corrected comments in instruction 2. A similar correction was made in Recipe 1, Step 6, although the method discussed there is not in fact used in the application (it is commented out).

- Recipe 1, Step 5.3, instruction 6. December 31, 2000. In earlier versions of `MyWindowController.m`, the `updateMyCheckbox:` notification method was registered as an observer with the notification center in the window controller's `init` method. Registering in the `init` method involved a subtle error, since the `[self mySettings]` object passed as the `object` parameter in the registration method had not yet been created and was therefore passed as `nil`. *Vermont Recipes* nevertheless functioned correctly, but it would not have functioned correctly later, when text fields are added to the application. The notification is now registered with the notification center in the window controller's `windowDidLoad` method.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 4:00 PM

Recipe 1: A simple, multi-document, multi-window application

Download the project files for Recipe 1 as a disk image and install them

Download a PDF version of Recipe 1

In the first Recipe, you will start by creating a new project in Project Builder, setting up the initial source files, nib files, and resources, as well as the correct folder structure for your project. You will then turn to Interface Builder to begin laying out the basic features of the application's graphical user interface. Finally, you will return to Project Builder to write code to complete the implementation of your initial user interface and to begin implementing the substantive functions of the application. When you have completed *Recipe 1*, you will have a working Cocoa application, with its own icons, double-clickable in the Finder and complete with an about box, support for multiple documents and windows, a tabbed view with user controls, multiple undo and redo, and the ability to save, open and revert documents.

The Vermont Recipes application is a document-based application relying on the Cocoa Application Kit framework. Like most Cocoa document-based applications, it adopts the so-called Model-View-Controller paradigm (MVC), which originated in the SmallTalk language from which the Objective-C extensions to C were derived. This is main-stream 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. If you haven't done so already, be sure to read about MVC in Apple's *Application Design for Scripting, Documents, and Undo*, a document installed on your computer at */Developer/Documentation/Cocoa/ProgrammingTopics/AppDesign* in both PDF and html format or downloadable as a PDF file.

A note about the Model-View-Controller paradigm

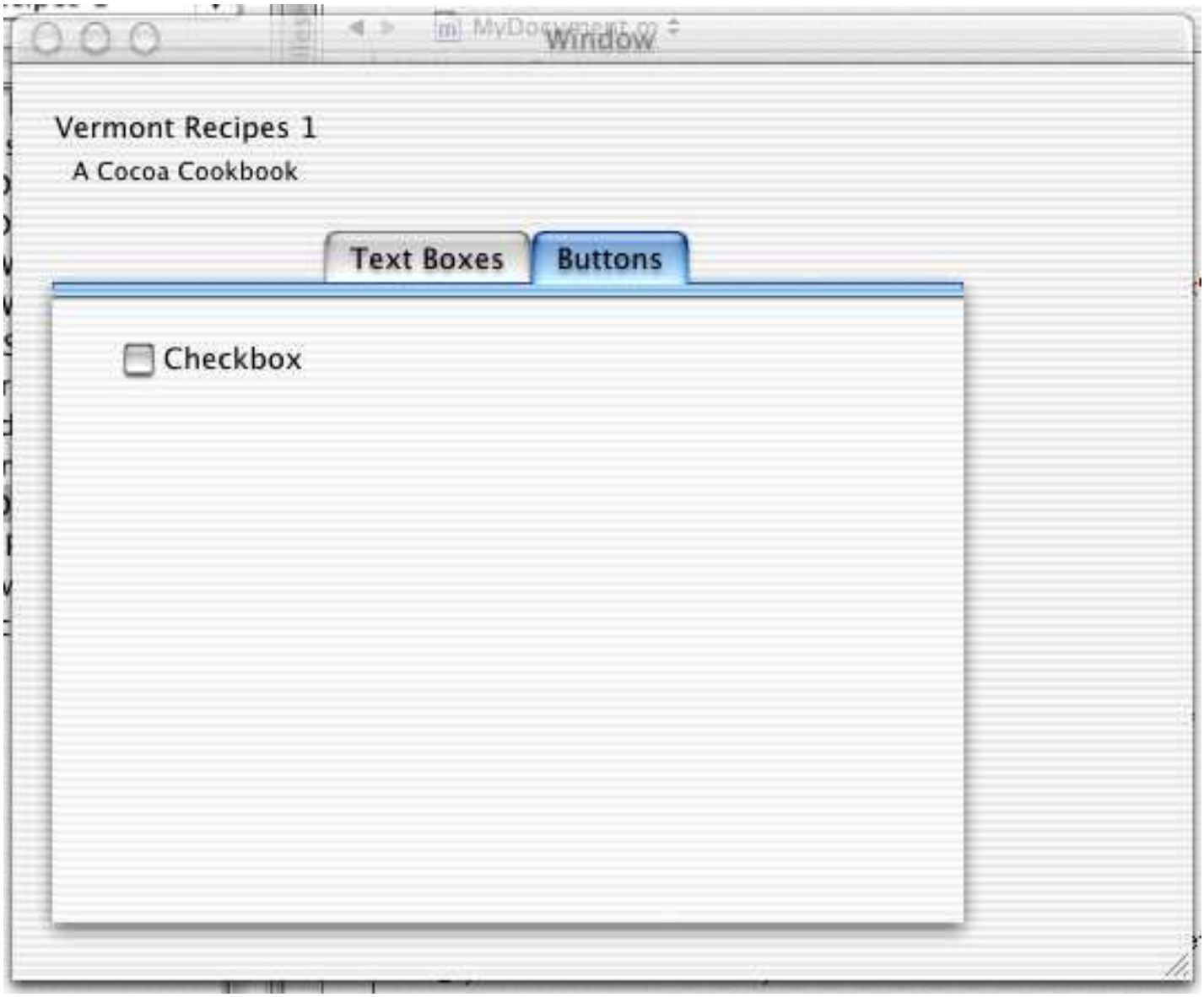
In the MVC paradigm, the "model" is where the application's data resides, including behaviors relating directly to the data. It is represented in a Cocoa application by one or more model objects, which are usually designed to be as independent of all other aspects of the application as possible. In particular, the code for the model should be completely independent of the user interface, having no knowledge of how any item of data in the model is represented on the screen. The NSDocument class, which you will subclass for the Vermont Recipes application, usually serves to manage these model objects and should not be thought of as a model, itself. The Cocoa frameworks, particularly NSDocument, already do much of the work for you, such as providing save and open sheets and the underpinnings of multiple undo and redo. You will be able to focus on writing the code that implements the application's unique data structures.

The "view" is where the graphical user interface exists. Most classes in the Cocoa AppKit are view classes, and they do most of the work of drawing and manipulating windows, panels and user controls for you. You will add views to the application using Interface Builder, requiring relatively little coding in Project Builder. What little code you might write for the views will be completely independent of the model's data structures, having no knowledge of how the data represented by a view is structured or where it exists.

The "controller" acts as an intermediary between the model and the view, allowing the application to maintain a complete separation between data and user interface. It is represented in Cocoa by the NSWindowController class, which you will subclass for Vermont Recipes and instantiate once for each window that the user opens. The application's views tell the window controller that the user has done something, and the window controller in turn tells the model to adjust its data accordingly. When the model's data is modified—for example, by an AppleScript command or the user's choosing the Undo or Redo menu item—it notifies the window controller, which in turn tells the affected views to update their visual state. The controller is where you must do most of your coding, since the interaction between the application's data and its user interface is unique and cannot be anticipated in the Cocoa frameworks. A simple Cocoa application can be written without subclassing NSWindowController, but the greater complexity of Vermont Recipes requires you to customize it.

There can be many controllers in a Cocoa application, and they aren't always identified by the term "controller" in the class name. For example, as suggested above, NSDocument can be thought of as a controller responsible for managing a document's model objects. Another that you may encounter in your reading is the built-in NSDocumentController class, which, among other things, manages some aspects of a document's relationship with the file system. Don't let these naming issues confuse you, but focus instead on the role that the Cocoa classes play in the overall structure of an application.

Screenshot 1: The Vermont Recipes 1 application



Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe01/recipe01.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

The name of Interface Builder, commonly referred to as "IB," suggests that it is a tool for building, not just designing, a graphical user interface (GUI). It is both, and much more. In the course of designing the GUI for the Vermont Recipes application, you will see that the nib file built by Interface Builder contains a significant amount of information that will be used by the application at run time. Unlike many GUI design utilities, which generate uncompiled code, and unlike ResEdit in the classic Mac OS, in which you create layout templates, Interface Builder allows you to create classes and objects and to archive them for loading directly into your application at run time.

Documentation

The best place to begin reading about Interface Builder is in IB Help in the Developer Help Center in the Apple Help Center. Also read the latest Interface Builder release notes, which appear automatically when you launch Interface Builder. Both IB Help and the release notes are available directly from Interface Builder's Help menu.

An example project with which to practice your skills is found on your computer at */Developer/Documentation/DeveloperTools/InterfaceBuilder*. *Learning Cocoa* (O'Reilly, 2001) contains extensive instruction on the use of Interface Builder in the context of this example.

The old NextStep/OpenStep *Tools & Techniques Book* contains detailed step-by-step instructions for using Interface Builder that are still valuable. The book is on your computer at

/Developer/Documentation/Cocoa/DevEnvGuide/Book/Tools&TechniquesBook.pdf.

For general information about Interface Builder and other developer tools, read *Developer Tools Overview* at */Developer/Documentation/DeveloperTools/DevToolsOverview.html*. A convenient roadmap appears at */Developer/Documentation/DeveloperTools/devtools.html*.

You can see some of the information contained in a nib file even now, after just setting up a Cocoa document-based application. Launch PropertyListEditor, in */Developer/Applications/*, and use it to open the `classes.nib` component of `MyDocument.nib`, which is in the `English.lproj` subfolder of the project bundle. You will see an expandable outline showing the Cocoa classes and related information already known to the nib file simply from setting up the project files. You will add more information to the nib file in this Step, but much of the information in a finished Cocoa nib file will be in the form of archived Cocoa classes that are not in XML format and cannot be read using PropertyListEditor.

When you reach the point of writing code for the application, you will discover why the nib file's internal information is important. Many seemingly essential items will be missing from the Project Builder source files. For example, some instance variables will be declared in code, referring to user control objects, yet there will appear to be no code telling the application which user control is pointed to by any of the instance variables. Similarly, some methods will be implemented in code, apparently telling the application what actions to take when particular user controls have been clicked, yet there will appear to be no code telling the application which action method will be invoked by any of the controls.

The missing items are supplied by Interface Builder's nib files, which the application will open when it is launched. As you design the GUI for the application, you create "outlets" and "actions" and connect them to appropriate objects using Interface Builder. Outlets are Interface Builder-speak for instance variables that can be connected to objects in the nib file—you use Interface Builder to draw a connection from an object containing an outlet to the associated user control, then specify which of the object's instance variables to connect. Actions are Interface Builder's term for action methods—again, you use Interface Builder to draw a connection from a user control to the target object that implements the control's action method, then specify which of the target's action methods is to be invoked when the user clicks that control. Cocoa will automatically invoke the correct method at run time whenever the user clicks the control, with little or no further coding on your part.

As you use Interface Builder to create these outlets, actions, and connections, the information is stored in the nib file. The nib file is an integral part of the application, and the information in it is used at run time to pull everything together.

A note about outlets and actions

You will see in this Step 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 objects having outlets and the objects that are to be plugged into them. Interface Builder even uses a little electric outlet symbol to identify outlets.

In programmer's terms, an outlet is an instance variable declared in a class interface. Typically, a document class or a window controller class declares numerous outlets identifying other objects with which it needs to communicate, including not only user control objects but any kind of object.

In *Step 2.6.1*, you will see how you can also wire user interface objects with a sort of control circuit, connecting each user control object to a specific action method in a target object, to be invoked when the user control is turned on or off. This implements what is known in computer science circles as the target-action design pattern, a pattern that lies at the heart of Cocoa's design. Typically, a user control is connected to an action method implemented in a controller object.

In this way, Interface Builder allows you to divorce your user interface code from your substantive code to a much greater extent than is true of other programming environments. One of Interface Builder's functions is to let your application know at run time what is connected to what, so that you don't have to lock this information into your code at compile time. Interface Builder is much more than its name suggests.

2.1 Create the main document window

It is convenient next to design and build the essential elements of the user interface for the application's main document window. To start, you must create the window and connect its wiring.

1. In Project Builder, expand the Resources folder in the left pane of the project window and double-click `MyDocument.nib`. Interface Builder launches and opens the nib file. Alternatively, you can launch Interface Builder and use it to open the nib file.

   Two windows and a palette appear: the main Interface Builder window, entitled "MyDocument.nib," which initially contains the File's Owner, First Responder, and Window icons; the main document window, initially called "Window" and containing a string reading "Your document contents here;" and the Cocoa objects palette, also know as the objects window, from which you will drag various user controls to the main document window. The palette is hidden whenever you bring another application to the front.

A second palette appears when you choose Tools > Show Info. This "Info" palette, also known as the "Inspector", changes its content when you select different windows or user controls in Interface Builder. You use the Info palette for many purposes, including to set a user control's default properties or attributes and to select targets for a variety of connections. Like the objects palette, it is hidden whenever you bring another application to the front.

You can change Interface Builder's preferences so that the Info palette opens automatically when the application is launched, and so that the objects palette and the Info palette appear as utility windows.

In case you're wondering, "nib" is said to stand for "NextStep Interface Builder."

2. Explore the default instances and classes that have been set up for you. The full meaning of what you find will become clear later.

    a. Choose tools > Show Info with the main document window selected. In the Window Info palette, use the pop-up menu at the top to select the Connections pane, if necessary, and click the delegate connection at the bottom. A line appears, extending from the main document window's title bar to the File's Owner icon in the `MyDocument.nib` window. This indicates that the Window object delegates some of its functionality to your `MyDocument.nib` file's owner. You will find the file's owner in a moment. Delegation is an important Cocoa concept; you will learn more about it later.

    b. Click the File's Owner icon in the `MyDocument.nib` window. The line disappears and the Info palette automatically switches to the File's Owner Info palette.

    c. Click the `window` connection in the connections area at the bottom of the Connections pane of the File's Owner Info palette. A line appears, extending from the File's Owner icon to the main document window's title bar. This indicates that the default file's owner in your project (whatever it is) has an outlet, or instance variable, called `window`, and that it points to an NSWindow object called "Window."

    d. Choose Attributes or Custom Class in the pop-up menu at the top of the File's Owner Info palette. A list of all available classes appears, with MyDocument selected at the top. This indicates that the "file's owner" of `MyDocument.nib` is a class called "MyDocument."

        You know it is a subclass of some other class, because it does not contain the "NS" prefix that identifies most built-in Cocoa classes. In fact, it is a subclass of the Cocoa NSDocument class, and it is declared in the `MyDocument.h` header file that you created using Project Builder in *Step 1*.

        According to an Apple insider, the original prefix for NextStep classes was "NX." This was changed to "NS," for NextStep, when the Foundation framework was made available, before the OpenStep protocols were released. Now, of course, it stands for the core Cocoa

frameworks, Foundation and AppKit. A prefix of some sort is recommended for all classes in publicly-distributed frameworks, in order to minimize the chance of global namespace collisions.

e. Click the Classes tab in the `MyDocument.nib` window. An outline list of all available classes appears. Most of them are dimmed, indicating that they are built-in Cocoa classes that you cannot edit.

f. Scroll down until the MyDocument class comes into view, if necessary. This is the same MyDocument class you encountered in the File's Owner Info palette in instruction d., above, where you learned that it is the file's owner. It is black, indicating that it is a custom class that you can edit.

   A shortcut to find a class in the Classes pane is to double-click the class's icon in the Instances pane of the main Interface Builder window. Try it. The window automatically switches to the Classes pane and shows the MyDocument class selected.

g. Click the MyDocument class in the Classes pane to select it, if necessary, then choose Classes > Edit Class. The MyDocument class entry in the outline expands to show two sublists, called Outlets and Actions, respectively. The dimmed entry under Outlets indicates that the MyDocument class, which you know is the file's owner, declares a default outlet called `window`. This is the same window you encountered in the File's Owner Info palette in instruction c., above, where you learned that it is connected to the Window object. Return to the full list of classes by clicking the disclosure triangle in the line above MyDocument to collapse the outline.

   A shortcut to see a class's outlets and actions is to click the outlet or the action button to the right of the class name in the Classes pane.

h. Click the Instances tab in the `MyDocument.nib` window. The absence of a MyDocument icon in the Instances pane indicates that the custom MyDocument class has not been instantiated. This is as it should be, because you want documents to be instantiated only when, for example, the user chooses File > New or File > Open while the application is running. For now, just remember that this is the way you can examine classes in your application even though they have not been instantiated.

i. To visit the final stop in your introductory Interface Builder tour, look at the little buttons that appear at the top of the vertical scroll bar in the `MyDocument.nib` window when the Instances pane is showing. The top button is the Icon Mode button. The second button is the Outline Mode button. Click the Outline Mode button. You see a list naming the three objects whose icons you saw while in Icon Mode. Click the disclosure triangle to the left of the "NSWindow (Window)" entry. You see the one NSTextField object currently showing in the document's main window, together with its contents ("Your document contents here"). Later, after you add more user controls, you will see them in the expanded outline, as well. Next, click one of the wedges that are enabled in the right pane of the `MyDocument.nib`

window. Lines appear showing all of the incoming and outgoing outlets relating to that object. Return now to the Instances pane of the window in preparation for the next instruction.

3. The *Vermont Recipes* application specification tells you that the application's main document windows will have drawers. Because of this, you must alter `MyDocument.nib` so that it has a window with a drawer, rather than the simple window provided by the Cocoa Document-based Application template's nib file. For the time being, you will leave the original Window object in the nib file to remind you of its connections, some of which must be duplicated in the new window object you will now create. You will not actually create the drawer until a later Recipe, but you can avoid a lot of extra work at that time if you set the stage for it now.

   If you plan to build an application based on *Vermont Recipes* but using a simple document window without drawers, just skip the instructions here that relate to creating a window with drawers. Everything in *Vermont Recipes* relating to the Parent Window will work the same way in a simple window.

   a. At the top of the objects palette, click the icon for the Windows palette in order to show its pane. Move the slider control to bring its icon into view, if necessary. The Windows palette icon portrays an empty window with a title bar and standard title bar buttons; it is the fourth icon from the left in a basic Developer Tools installation that does not include custom palettes.

      If in doubt about which icon represents the palette you want, just hold the mouse pointer over an icon for a moment. A Help tag appears displaying its name. Then you can move the mouse pointer from icon to icon without pausing, in order to see the Help tag for each in turn.

   b. Drag the icon representing a window with an open drawer and drop it into the `MyDocument.nib` window. Three new icons appear: an "NSDrawer" icon, a "DrawContentView" icon, and a "Parent Window" icon. A new, empty window entitled "Window" and a new panel entitled "DrawContentView" also appear on the desktop.

   c. You can determine which window or panel is associated with which icon in the `MyDocument.nib` window by double-clicking an icon to bring its associated desktop window or panel to the front.

   d. Using this technique, bring the Parent Window to the front. It is a little too small for your purposes, so drag the lower right corner to make it about an inch wider and taller.

4. Explore the new objects you just created.

   a. Click the NSDrawer object icon to select it. In the NSDrawer Info palette's Connections pane, you see that two outlets, `contentView` and `parentWindow`, are already

connected, the former to an NSView object and the latter to an NSWindow object.

b. Click the `contentView` connection in the connections area at the bottom of the NSDrawer Info palette. A line appears, extending from the NSDrawer icon to the DrawContentView icon in the `MyDocument.nib` window. This indicates that it is the content area to which the outlet is connected; drawers do not, of course, have title bars.

c. Click the `parentWindow` outlet connection in the NSDrawer Info palette. A line appears, extending from the NSDrawer icon to the title bar of the Parent Window.

d. The DrawContentView and Parent Window icons have no connections at this point, although they do have outlets. You saw above that the original window had a connection from its delegate outlet to the MyDocument class, but you will not duplicate that connection in the parent window because you will create a different delegate connection in a later Step.

5. To be consistent with Aqua human interface guidelines, a new, untitled window should open when the application is launched and, of course, whenever the user opens a new document. An untitled window should also open when the application is already running and is brought to the front, for example, by clicking its icon in the Dock, unless a window owned by the application is already open (whether or not minimized). Now is a good time to take care of this detail. Click the new Parent Window icon in the `MyDocument.nib` window to select it, then select the Attributes panel of the Window Info palette from the pop-up menu at the top. Click the Visible at launch time checkbox. A check mark appears.

   This Interface Builder attribute setting will cause Cocoa to show the main document window automatically every time a new document is created and loads its nib file. If you did not check this setting in Interface Builder, you would have to invoke a method somewhere in your source files to show the window. Eventually, you may discover that it is preferable to do it in code, anyway, for better control over the time when the new window opens or to provide a user preference setting to turn this feature off. For a window that has complex content, using the attribute setting in Interface Builder may allow the user to see the contents as they update, instead of presenting them all at once, fully configured, for a more polished appearance. For now, however, setting an attribute in Interface Builder is a convenient means to set default application behavior without writing code, and you will use it frequently while creating Cocoa applications.

   Notice that this attribute only tells a newly opened document and nib file to show its main window. If you wonder how the new document itself gets created when the application is first launched, examine the documentation for the NSDocumentController class, where you will learn that this behavior is hard-wired in Cocoa. One of the conveniences of Cocoa is that standard Macintosh behavior like this is often provided automatically by built-in Cocoa classes and methods. There are usually other methods, parameters and Interface Builder attributes available to modify the standard behavior, if desired.

   In general, the Info palettes in Interface Builder give you a lot of control over initial attributes of objects without requiring you to set them explicitly in code. For example, if you switch to the Size

pane of the Window Info palette, you see that you can set the initial size and location of the window, minimum and maximum size constraints, and how controls within the window respond to resizing the window. You will explore these and other important attributes in later Recipes.

6. You must change the connection between the file's owner and its window object, because it is currently a connection to the original simple window created by the Cocoa Document-based Application template. You will soon discard this simple window.

    a. Click the File's Owner icon to select it, then choose the Connections pane of the File's Owner Info palette from the pop-up menu at the top.

    b. Click the `window` connection in the connections area at the bottom of the File's Owner Info palette. You see, as before, that its `window` outlet is connected to the title bar of the original window. The Disconnect button is enabled.

    c. Click the Disconnect button. The connection is broken.

    d. Hold down the Control key and drag from the File's Owner icon to the Parent Window icon in the `MyDocument.nib` window. A line is drawn from the File's Owner icon to the Parent Window icon as you drag. When you let up the mouse button, the `window` outlet in the connections area of the File's Owner Info palette is selected, and the Connect button is enabled.

    e. Click the Connect button. The new connection appears in the connections area at the bottom of the File's Owner Info palette. Click in a blank area of the `MyDocument.nib` window to make the line disappear, if necessary, then click the File's Owner icon again and click the new connection again in the File's Owner Info palette. This time, a line appears from the File's Owner icon to the title bar of the window associated with the Parent Window icon.

7. Now you can delete the original Window object provided by the Cocoa Document-based Application template. Click the Window icon (not the Parent Window icon) in the `MyDocument.nib` window to select it, then choose Edit > Delete or hit the Delete key. The Window icon and the original main document window disappear. No confirmation dialog was presented to let you cancel this action, but you can undo it by choosing Edit > Undo Delete.

8. Save your work in the nib file by choosing File > Save.

In the next Step, you will insert some text, a tab view and a checkbox user control into the parent window and set them up.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.2 Add a tab view and user controls to the main document window

The application specification tells you that the application's main document window will contain a tab view, to permit the user to switch among multiple panes within the one window. The window also needs several controls to let the user edit the document's settings and data. In this Step, you will create some static text in the upper left corner of the main document window to identify the document, a tab view filling most of the rest of the window, and a single user control, a checkbox, in the second pane. In later Steps you will add additional tabbed panes and user controls using the same techniques.

A note about tab views

Tab views have become a common element of Macintosh user interface design. They are particularly suitable in Mac OS X, because Mac OS X places increased emphasis on minimizing the multi-windowed desktop clutter characteristic of Mac OS 9 and earlier. Using tab views is an intuitive and efficient way to increase the amount of information that can usefully be contained within a single window.

It is somewhat easier to include a tab view at the outset than it is to add one later, so you will start by creating a tab view here. You will discover that using a tab view adds almost no complexity to a Cocoa application's code, because the built-in NSTabView and NSTabViewItem Cocoa classes do almost all of the work. In fact, the application's code would be almost identical in an application free of tab views.

Part of the reason for this is that a Cocoa document window without a tab view is not without any view at all. In fact, there is an implicit view in every window to hold its contents. That implicit content view performs many of the functions that an explicit tab view or other view performs, particularly with respect to embedded user controls.

Start where you left off in *Step 2.1*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

1. In the `MyDocument.nib` window, double-click the Parent Window icon. The empty document window associated with the Parent Window icon comes to the front.

2. The main document window needs some static text to tell the user what it is.

    a. In the objects palette, select the Views palette. Its icon portrays a push button and a text field containing the word "Text"; it is the second icon from the left in a basic Developer Tools installation without custom palettes.

    b. Drag the "Message Text" NSTextField icon from the palette and drop it in the upper left corner of the empty main document window. When you drag it near enough to the upper left corner, vertical and horizontal guidelines appear. Drop it while the guidelines are visible to ensure that it is placed in accordance with Aqua Human Interface Guidelines. This is an ordinary NSTextField view with its font and font size preset to comply with Aqua human interface guidelines for message text, but it is also suitable as is for the use you will make of it here. It is surrounded by selection knobs indicating that the NSTextField object is selected and can be resized or dragged to a new position.

    c. You must edit the text field's content. If the text field has become deselected because you clicked outside of it after dropping it in the window, reselect it by clicking the text once. Then select the text within the text field ("Message Text") for editing by double-clicking to highlight it. Type **`Vermont Recipes 1, My First Cocoa Application`** over the selected text. If the text field does not expand to accommodate the new contents as you type, you may have to drag one of its selection knobs to enlarge it.

    Suppose you now decide the name is a little trite. Select the text for editing again, if necessary, and delete everything after "Vermont Recipes 1". If the field does not contract after you delete the text, you can resize the field again manually by dragging any of its knobs while the field (not its text) is selected.

    As an alternative to resizing a user control by dragging its selection handles, you can resize it to precisely contain its content by choosing Layout > Size to Fit while either the field or the content is selected. Try this now, after changing the control for testing purposes. You see the border and handles demarking the object move until they just fit around the text. Click on an empty area of the window to deselect the NSTextField object.

d. If you were to reposition the text field, perhaps on the theory that it serves as a sort of logo for this document and therefore need not comply with the Aqua human interface guidelines for user controls, you would simply click the text to reselect the field (not its content) and drag it to its final location near the upper left corner of the window. You can click and drag in one motion.

e. Next, examine its attributes. While the NSTextField object is selected, click the NSTextField Info palette to bring it to the front, then select the Atttributes pane using the pop-up menu at the top. Make sure the Editable and Selectable checkboxes in the Options area are unchecked, since this text should not be changed by the user and there is no need to allow the user to select and copy it.

f. Using the techniques you just learned in instructions a.-e., above, use the "Informational Text" icon in the Views palette to create another static text field immediately below the first, and edit it to read **`A Cocoa Cookbook`**. Drag as necessary to position it close beneath "Vermont Recipes 1," relying on the guidelines for its exact position. To center it beneath "Vermont Recipes 1," select the top text field, Shift-click the bottom text field to add it to the selection, and choose Layout > Alignment > Align Vertical Centers. Don't forget to choose the Attributes pane of the NSTextField Info palette while the lower text field alone is selected and change its attributes to the same settings you used in instruction e., above.

3. It is now time to install a tab view in the Parent Window.

   If you plan to adapt the *Vermont Recipes* approach to an application that does not utilize tab views, just skip the instructions that relate to tab views. Almost everything else in *Vermont Recipes* will remain applicable, without change.

   a. In the objects palette, select the Tabulation Views palette. Its icon portrays an empty, scrollable table view with two column headings; it is the sixth icon from the left in a basic Developer Tools installation.

   b. Drag the NSTabView icon from the palette and drop it in the main document window immediately below the text fields you just added. Resize the tab view by dragging its bottom right selection knob almost to the bottom of the window and to a spot about one inch from the window's right edge. Guidelines will appear to help you place it, but in this case you should ignore the vertical guideline on the right because you want to leave room to add user controls near the right edge of the window later.

   c. There are currently two tabs on the tab view, each representing a separate pane, or tab view item object, within the tab view object. Neither tab has the title you want. To change the text of a tab's title, start by clicking once anywhere in the tab view to select it, then double-click anywhere in the tab view to select the pane, or tab view item, having a highlighted tab; a broad border in your highlight color appears around the tab view, superimposed on its selection knobs. Then, if the first tab is not highlighted, click once on the first tab to select

the tab view item associated with it. Finally, double-click on the text in the first tab to create a text entry area where you can type its title. Type **Text Boxes**. Then deselect the text entry area by hitting the Enter key or clicking outside of the tab view. The tab widens automatically to fit the text.

You encountered the requirement to select a control first and then separately double-click to select its content for editing once before, in instruction 2.c., above. In a future version of Interface Builder, you will not have to go through this somewhat laborious process.

d. Using the technique you just learned in instruction c., above, type **Buttons** on the second tab. Leave the Buttons tab highlighted for the next instruction.

4. Assume the application specification requires that the main document window contain a switch, or checkbox, user control.

    a. Select the second pane, or tab view item object, in the main document window's tab view, if necessary, by double-clicking anywhere in the tab view to select the pane area and clicking the second tab to select the Buttons pane. It is important that the desired tab view item object remain selected as you carry out this instruction 4 (the entire tab view should be surrounded by a broad line in your highlight color, indicating that a tab view item is selected, and the correct tab should be highlighted). Otherwise, the checkbox control will land in the tab view when you drag and drop it from the objects palette, instead of landing in the desired tab view item, and it will remain visible when you switch to a different pane.

    b. Select the Views palette.

    c. Drag the Switch NSButton object icon from the Views palette and drop it into the Buttons pane in the main document window's tab view, first making sure that the Buttons pane, or tab view item, is still selected (it has a broad line around it). Place it near the upper left corner of the tab view item, where the guidelines indicate.

    d. You must edit the control's label. Instead of using the double-click technique you learned in instruction 2.c., above, you can instead, if you prefer, use the NSButton Info palette as an alternative means to edit the label. While the Switch control is selected, click the NSButton Info palette to bring it to the front. In the Attributes pane, edit the Title ("Switch") to read **Checkbox**. When you tab or click out of the Title field or hit the Enter key, the label of the checkbox in the main document window changes to "Checkbox". If the control does not enlarge automatically to show the new title in its entirety, choose Layout > Size to Fit while the control is selected.

    e. Help tags, sometimes called Tool tips, are cool. With the Checkbox control selected, select the Help pane of the NSButton Info palette. In the Tool Tip box, type **Toggle Checkbox** and hit the Enter key. You will provide additional help in a later Recipe.

5. You can test the document window at any time to be sure the controls are working. Choose File >

Test Interface. The Interface Builder windows and palette disappear, leaving only your document's main window, as if you were actually running the completed application. Click once in the main document window to make sure it is frontmost. Click on the two tabs alternately to see that one and then the other pane appears, with only the Buttons pane showing the Checkbox control. (If both panes show the control, you neglected to keep the Buttons pane selected when you dragged the Switch control to it in instruction 4.c., above.) Click repeatedly on the Checkbox control in the Buttons pane to see that a check mark appears and disappears. Leave the mouse pointer over the Checkbox control for a moment to see that the Help tag appears; when you move the pointer away, the Help tag slowly fades away. To return to Interface Builder, choose Interface Builder > Quit Interface Builder to terminate the interface testing mode. The Interface Builder windows and palette reappear.

6. Save your work in the nib file by choosing File > Save.

In the next Step, you will use Interface Builder to create a window controller class to act as an intermediary between the application's data and its user interface.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe01/recipe01_step02_02.html

Introduction > Contents > Recipe 1 > Step 2.2 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.3 Create a subclass of the window controller class

You know from the *Vermont Recipes* application specification that this application will allow many documents to be open at once, each supporting more than one kind of window. Each kind of window will have its own nib file archiving its unique set of user controls and connections, such as the main window you created in *Step 2.1*. According to the Cocoa documentation for document-based applications, you will need a window controller for each kind of window the application's documents can create. Since these window controllers will perform many functions unique to the Vermont Recipes application, you should subclass NSWindowController once for each kind of window. In this Step, you will create an NSWindowController subclass for the application's main window.

A note about the window controller class

In a document-based Cocoa application using the MVC paradigm, the window controller class is especially important. Before proceeding with this Step, be sure to read the *NSDocument* and *NSWindowController* class reference documents in the Developer Help Center, and pay particular attention to the *Document-Based Application Architecture* section of the *NSDocument* class reference. See the following note on Documentation to learn how to find Cocoa class reference documents.

In a sense, the window controller is the least standardized of the model, view, and controller classes, because it must tell the model and the views how to perform the unique functions of the application and keep them synchronized. It manages a window of the

document on behalf of the document. A model class representing the application's data should know nothing about the graphical user interface, and the view classes, such as windows, tab views and user controls, should know nothing about the specific data that they represent. You don't have to subclass NSWindowController for a very simple application, but you usually do for a real world application.

Only the controller class knows how the model and the views interact. The document tells the controller when data has changed (perhaps the user has reverted to the document's saved state or chosen Undo or Redo), and the controller then tells the views to change state to reflect the new data. Similarly, the views tell the controller when the user has changed the state of a user control (perhaps the user has clicked on a checkbox control), and the controller then tells the model to update its data stores accordingly. For these reasons, you will likely write much more custom code for the controller than you do for the model or the views.

Modern computer science 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 upgrades.

But there are also specific reasons relating to the way the Macintosh works. In particular, the graphical user interface is not the only user 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 user control. In doing so, AppleScript does not need to know anything about the graphical user interface but can communicate directly with the document's model objects. By keeping the model separate from the views and relying on the controller to mediate between them, Cocoa's AppKit is able to give you AppleScript support almost for free. Just as the user's reverting a document to its saved state causes the model to tell the window controller to update any affected views, so a script's alteration of the model's data also, by invoking the same methods, causes the document to tell the window controller to update any affected views.

In addition, in the Cocoa environment, many classes perform their work by invoking so-called "delegate methods," which you as application designer can choose to implement or not to implement in your own subclasses. This is one of the ways in which Cocoa is able 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.

Documentation

The Developer Help Center in the Apple Help Center is a convenient place to read the detailed class reference documentation that is provided with the Developer Tools, such as the *NSDocument* class reference referred to in the previous Note. In the Developer Help Center, click on Cocoa, then click either on Application Kit or Foundation to find an index of all of the class references. Don't overlook the function, protocol and other references near the bottom.

The class reference documents are an extremely important source of information about the proper way to use the Cocoa classes. Along with the comments often included in the class header files, the class reference documents may be the only source of information available to guide you in designing and coding important features of a Cocoa application. You should not use any features of a Cocoa class without first becoming familiar with its class reference and, in many cases, its header file.

The class reference documents can also be found on your computer both in HTML and PDF format. The actual location of a class reference is deep in the System Library's `Frameworks` folder. The current documentation for an AppKit class, for example, is located in the most recent subfolder of the `Versions` folder in `AppKit.framework`; at this writing, that is the `C` subfolder. You can be assured of reading the most recent version by opening the `Current` subfolder, at the same level, which is actually a symbolic link to the most recent subfolder. Within the most recent subfolder, the header files are in the `Headers` folder and the class reference documents are located deep in the `Resources` folder, in the `Documentation` subfolder of the `English.lproj` folder. The path to the HTML version of the current *NSDocument* class reference at this writing, for example, is the following:

/System/Library/Frameworks/AppKit.framework/Versions/C/Resources/English.lproj/Documentation/Reference/Obj-C_classic/Classes/NSDocument.html

A much more convenient shortcut to find the HTML and PDF versions of the class reference documents is this:

/Developer/Documentation/Cocoa/Reference/ApplicationKit/Obj-C_classic/

Start where you left off in *Step 2.2*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

1. In the `MyDocument.nib` window, select the Classes tab. An outline list view of all available classes appears.

2. Scroll to the bottom of the list, if necessary, and click once on the NSWindowController class to select it. The line on which NSWindowController appears is highlighted.

3. Choose Classes > Subclass. A new subclass named "MyWindowController" appears on a new line, indented under NSWindowController. Its text is black, not gray, to indicate that it is a custom class which you can edit, and it is selected.

4. You can rename the MyWindowController subclass by double-clicking its name and typing over it. But "MyWindowController" is perfectly descriptive, so accept it as is. If its text is selected for editing, hit the Enter key to deselect it.

5. At this point in the process of creating a new subclass in Interface Builder, it is possible to instantiate an object based on the new subclass. However, you should not instantiate the MyWindowController subclass at this time, because this is to be a multi-window, multi-document application. You will instead want a new instance of the window controller to be created programmatically when a new, empty document is created when the application is launched and again each time the user requests a new document at run time. Later, in *Step 3.3*, you will write code so that your application can create window controller instances at appropriate times.

   Interface Builder nevertheless needs to know about the class because later, when the time comes to define connections among MywindowController, its main window, and its window's user controls, you must be able to designate the MyWindowController subclass as the "owner" of `MyDocument.nib`. The File's Owner icon in Interface Builder is a sort of proxy or stand-in for a class that is not yet instantiated, allowing you to use Interface Builder to create connections between that class and other objects.

6. Save your work in the nib file by choosing File > Save.

In the next Step, you will use Interface Builder to confirm that your nib file already has a document class.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.4 Confirm that you have a subclass of the document class

The user's work in your application's main document window must eventually be saved to nonvolatile, or persistent, storage, such as a hard disk, so that it can be recovered and used at a later time or by other users. Live data is maintained in an application's internal data structures in RAM while a user is editing it, and it is saved to storage in a specific file format periodically and when the associated document is closed. The word "document" is used to refer both to the file on disk and to its live data representation in RAM. "Document" in this sense is not to be confused with the NSDocument class, which is a controller rather than a repository of data.

According to the Cocoa documentation, a subclass of Cocoa's NSDocument class is required so that a new instance of it can be created with appropriate elements and attributes whenever the user creates a new document or opens an existing document from storage. The document subclass has already been provided to you in the Cocoa Document-based Application template files `MyDocument.h` and `MyDocument.m`, as you saw in *Step 1*, but it must have a corresponding entry in `MyDocument.nib` so that appropriate outlets can be created.

A note about the document class

Every document-based Cocoa application must subclass NSDocument. As was mentioned in *Step 2.3*, the document subclass controls the application's data model. It is considered a controller class in the MVC paradigm. You must subclass NSDocument, among other reasons, to provide access to data variables in model objects that hold your document's data, to provide accessor methods to enable other objects to get and set the data values, and to provide methods to tell the window controller when the data has changed so that the 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 scripting.

Cocoa also has an NSDocumentController class, but it is almost never necessary to subclass it. NSDocumentController handles document-related behaviors that are common to all document-based applications, such as opening documents when needed and reporting on various properties of the application. When customization of NSDocumentController is required, it is usually preferable to use an application delegate.

Start where you left off in *Step 2.3*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

1. In the `MyDocument.nib` window, select the Classes tab. An outline list view of all available classes appears.

2. The Cocoa Document-based Application template has already provided you with an NSDocument subclass named "MyDocument." Scroll about a quarter of the way down the Classes outline view until you find the MyDocument class, indented under the `NSDocument` superclass.

   You should not instantiate the MyDocument class in Interface Builder, because a new instance of the document must be created each time the user requests a new document at run time in a multi-document application. Cocoa will create a connection between the new document and its associated window controller so that your window controller can talk to the document, for example, to set or get the data it controls.

3. If you wanted to rename it, you could now double-click the name of the MyDocument class to select it for editing, and rename it by typing. However, giving it the same name as the header and source files will avoid confusion, so you should leave it as you found it.

4. Save your work in the nib file by choosing File > Save.

In the next Step, you will use Interface Builder to designate the owner of the nib file.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.5 Designate the owner of the nib file

`MyDocument.nib` needs an "owner" to take responsibility for loading it into memory when a document and its main window are opened. The Cocoa Document-based Application template includes a nib file that makes the MyDocument subclass the default owner of `MyDocument.nib`, because the template is designed for a simple application having only one kind of document represented in a single kind of window. However, the Cocoa documentation indicates that you should normally create multiple MyWindowController subclasses in cases where a document will have multiple kinds of windows, or where you need to customize the built-in behavior of NSWindowController. It is convenient to make the MyWindowController subclass the owner of `MyDocument.nib` in such an application, to facilitate a one-to-one correspondence between the main document window and its unique window controller. Later, when you create other kinds of windows for the document, each will be given its own nib file and another unique window controller to manage it.

A note about the File's Owner

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

A proxy is used because in a document-based application the nib file's owner cannot be instantiated in the nib file, which is created while you are designing the application. As discussed in *Step 2.3*, the document and its windows must instead be instantiated programmatically at run time, when a new document is created. The owner of the nib file must exist in memory before it can load the nib file. The owner is in this sense external to the nib file that archives the window class.

However, a means of communication between the owner and the window defined in the nib file must be provided when you design the application. The File's Owner stands in for the owning object for this purpose, that is, to enable you to draw the necessary connections between it and other objects when you are designing the program.

Start where you left off in *Step 2.4*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

1. Click the File's Owner icon in the Instances pane of the `MyDocument.nib` window once to select it. The File's Owner Info palette appears.
2. Select the Attributes pane of the File's Owner Info palette. The MyDocument class is already selected as the default owner. Click on MyWindowController, just below MyDocument in the Class list, to select MyWindowController as the new owner of `MyDocument.nib`. If an alert appears, warning you that this step will break existing connections, click the OK button because you want to break any existing connections between objects in `MyDocument.nib` and its default owner, MyDocument.
3. Save your work in the nib file by choosing File > Save.

In the next Step, you will use Interface Builder to create outlets and actions and connect them to objects.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

3.5 Set up a model class to hold the application's data

In *Step 2.6.1*., you used Interface Builder to create an action to be invoked every time the user clicks on the checkbox. When you subsequently used Interface Builder's Classes > Create Files… command to generate the source files for the project, it created a stub method for this action in MyWindowController. Interface Builder left it to you to provide the code to make this stub method work, however. Since the user's clicking on the checkbox should initiate a change in the state of the application that the application will remember, you must provide some architecture that the application can use to store data. In short, you are now forced to decide on a data representation for the application.

This is a very big step, and it will take you on a long digression before you can finally return to implement the action method in *Step 3.6*. Decisions you make now about how to represent settings and other data in the application will have a significant effect on the ease with which additional settings can be implemented and on the efficiency of the application itself. In normal software development, this is one of the most important elements of the application specification. A lot of false starts are avoided during coding if you have settled in advance on a data representation that will meet all of the application's eventual needs.

In general, the MVC paradigm contemplates that an application's data will be encapsulated in one or more separate model objects, each devoted solely to the representation and behavior of the application's data or some subset of it. You will therefore now create a separate class to hold the data value that is set and reset from time to time when the user clicks the checkbox control. This new class can be given any name. You will call it MySettings, here, implying that this model object will hold basic application settings. In later recipes, you will add variables to it, and you will create additional model objects to hold other subsets of

application data.

You will create a simple Boolean variable in the MySettings model object to hold the checkbox data value, along with a few methods by which other objects, such as the window controller, can access it. In the course of doing this, you will have to arrange to instantiate and initialize the new model object. Then, in the remainder of *Recipe 1*, you will implement all the additional elements of the application needed to make this design work, such as finally filling in the action method, and writing routines to undo and redo changes to the checkbox value, to save the state of the checkbox value to disk and read it back in, and to revert a changed checkbox value to its saved state. Later, in *Recipe 2*, you will add additional user controls using the same approach.

A note about data representation and model classes

The cardinal rule of data representation in document-based Cocoa application development is that you should represent the data in a model object.

Among other things, this means that you should *not* write an application in such a way that it depends upon the current visual state of a user control as a record of the state of the application's data. It may seem that reliance on whether, for example, a checkbox is checked or not is the most efficient way to know whether the setting represented by the checkbox is true or false. After all, Cocoa already contains routines that allow you to read the current visual state of the checkbox, so why not keep things simple by taking advantage of work already done? Indeed, conventional Mac applications sometimes use this technique, at least temporarily while, for example, a dialog is being presented for user interaction.

The chief practical problem with such a shortcut is that a typical application has many interfaces, several of which may independently have the ability to change the state of the setting reflected by the checkbox. For example, you may want to add a menu command as an alternative interface to let the user change the underlying data. Or you may (as you should) include AppleScript support in your application, and a user may run a script that changes the data. In either of these cases, you would have to write additional code to change the visual state of the checkbox to reflect the data change made by the menu command or the AppleScript command. The AppleScript issue is particularly serious, since scripts often want to access an application's data without having to waste time dealing with its graphical user interface.

Centralizing the representation of the data in one place makes it far easier to keep all these interfaces synchronized. It even makes it easier to add new functionality like saving and reading the data, undo and redo, and revert. It also pays dividends in code maintenance in the future, when you might want to add entirely new functionality related to the existing data or to add completely different data sets with their own functionality.

Note that the document object in Cocoa is not a model object, as you might think. Instead, the document object is a controller object devoted to controlling the application's data in a larger sense. For example, you will shortly place the application's generic, abstracted methods for storing and retrieving data into the document object, MyDocument. Not all controllers in Cocoa include "controller" in their names, although you might think that the function of NSDocument would have been clearer if it were named "NSDataController." Also, don't be confused by the fact that there is a separate class in Cocoa called NSDocumentController; that, too, is a controller class, but it controls documents within the context of an application, for example, the opening and closing of documents.

You should not, therefore, as you might initially assume, place specific data definitions and accessor methods in the document class, but instead in a separate class or classes devoted specifically to that purpose. You will now deal with the details of how individual data items are represented in the application by creating this new model class.

Launch Project Builder and open the project, if necessary.

3.5.1 Create a new model class using Project Builder

In *Step 1* and *Step 2*, you created files for two new classes, first, MyDocument from a Project Builder template while creating a new project and, second, MyWindowController as a file created by Interface Builder. Now, you will create a file for a new class from scratch in Project Builder. The process is very similar to what you did in *Step 3.1.4* to create the `Localizable.strings` file.

1. Choose File > New File… in Project Builder.
2. Select the Objective-C Class template under the "Cocoa" heading and click the Next button.
3. Set the name of the file to **`MySettings.m`** and check the checkbox to also create "MySettings.h".
4. Click the "set…" button to open a navigation sheet and, if necessary, click on the Vermont Recipes 1 folder to set the new files' location to the Vermont Recipes 1 project folder, and click the Choose button.
5. Click the Finish button.
6. In the Groups & Files pane, drag the new `MySettings.h` and `MySettings.m` file icons into the Classes folder, if necessary.
7. Click `MySettings.h` to bring the new header file into the right pane and, if you wish, type over the comments at the top provided by the template to identify this as a "// Vermont Recipes 1" file. Do the same with `MySettings.m`.

8. In `MySettings.h`, replace "#import <Foundation/Foundation.h>" with the following:

```
#import <Cocoa/Cocoa.h>
```

9. Choose File > Save.

3.5.2 Initialize the model object

Like most new objects, a MySettings object requires initialization beyond what is provided by the NSObject class from which it inherits. It will have two initialization methods, one of which will be its so-called "designated initializer," the initializer that should normally be invoked when a new MySettings object is created. It is common for Objective-C classes to have more than one intializer, although the basic `init` method is always provided as a way to initialize values to default values such as `nil`. It is a good idea to get object initialization routines in place early in the process of writing a new class.

A note about object initialization and the designated initializer

Initialization of Cocoa objects is governed by a well-defined programming convention that all Cocoa applications must follow. The convention is best described in the class documentation for NSObject, a fundamental class declared in the Foundation framework, in the section documenting NSObject's `init` method; you are urged to read it. Since most Cocoa classes inherit from NSObject (and most of those that don't nevertheless guarantee to follow 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 `self`. This `init` method is available in every object that inherits from NSObject. Most such classes override the `init` method, or provide one or more substitutes, 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.

In order to assure that the initialization methods of classes intermediate between NSObject and the class are called, one of the initialization methods of the class must begin by invoking an appropriate initialization method of its immediate super class. This is the designated initializer. If that 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.

Other initializers are often provided in custom classes for special purposes. Each of them must call the class's designated initializer, directly or indirectly, through a message to `self`, passing default values to its arguments. This guarantees that the variables of the class are initialized to default 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, that is capable of most completely setting up the initial state of the object—is usually the designated initializer. In most circumstances, an application will find it convenient to initialize a new instance of the class by calling the designated initializer. It is the developer's obligation to identify the designated initializer by a comment in the header file, so that clients of the class can know which initialization method to call.

The designated initializer for MySettings will be used by the document object that creates a settings object to pass in a reference to the document's self. This is a common technique in Objective-C and other object-oriented languages, to give a newly created object the ability to communicate back to the object that created it. You will see later that this back reference is needed, among other things, so that the settings object can tell the document object that created it that some data has changed, and the document object can then pass the news along to the window controller and, eventually, to affected user controls. You will also learn that a link from MySettings to MyDocument is needed so that the settings object can share the document object's undo manager.

1. In the header file `MySettings.h`, add the following instance variables between the braces after `@interface MySettings`:

```
@private
MyDocument *myDocument;
```

2. Before `@interface MySettings`, add this:

```
@class MyDocument;
```

   This is needed so the compiler will know you intended to use the MyDocument type in the header and didn't just do so by mistake.

3. Add the following method declaration after the `@interface MySettings` block :

```
// Initialization

- (id)initWithDocument:(MyDocument *)document; //
designated initializer
```

A simple `init` method will be defined shortly in the source file, but in accordance with our practice it is not declared here, in the header file, because it is an override method. The `initWithDocument:` method is a custom method and must be declared. It could have been written with the same name, `init:`. This would not make it an "overloaded" method in the sense used in other programming languages, because, although it would have the same name as the first, `init`, it would add a parameter, indicated by the colon. Objective-C would recognize that they are two separate methods by the presence or absence of the parameter. It is customary in Cocoa, however, for separate initialization methods to be given different names. Usually, additional initialization methods are named "initWith" followed by a name suggesting the additional parameter's type, here, "Document".

This method is called a "designated initializer," which means that other objects should normally call it when instantiating a MySettings object. It is guaranteed to call the initialization methods of all classes from which it inherits. In this way, a newly-created MySettings object will always be correctly set up and will always have the information it needs to locate its associated document. As a matter of practice, you should always include a comment indicating which method is the designated initializer.

4. In the source file `MySettings.m`, add the following initialization methods after `@implementation MySettings`:

```
//Initialization

- (id)init {
    return [self initWithDocument:nil];
}

- (id)initWithDocument:(MyDocument *)document {
    if (self = [super init]) {
        myDocument = document;
    }
    return self;
}
```

The simple `init` method invokes the designated initializer, passing `nil` as its parameter. The designated initializer will be invoked by the document object when it instantiates a new settings object and initializes it by passing in the document object.

This method utilizes the standard Objective-C technique that you first saw in *Step 3.3* for initializing a subclass of another class. Notice again the tricky test in the first line.

5. In the header file `MySettings.h`, add the following accessor method declaration, after the Initialization section:

```
// Accessor methods and conveniences

- (MyDocument *)myDocument;
```

6. In the source file `MySettings.m`, add the following accessor method definition:

```
// Accessor methods and conveniences

- (MyDocument *)myDocument {
    return myDocument;
}
```

7. In `MySettings.m`, also add the following after `#import "MySettings.h"`, so that it will compile:

```
#import "MyDocument.h"
```

3.5.3 Enable a document to instantiate a model object and access its data

MySettings is now a declared class in the project, but nobody yet instantiates an object of this class. A MySettings object will serve strictly to hold and manipulate data associated with the document, so a document object is the obvious candidate to fill this role. Every document will have one and only one associated MySettings object, and the MySettings object will exist for the life of the document.

You will therefore arrange for MyDocument to instantiate a MySettings object immediately when any document is instantiated, and to release the MySettings object when its associated document is released. You will create an instance variable and accessor method in MyDocument so that the document can access its settings.

1. You must create an instance variable in MyDocument to hold a MySettings object. In `MyDocument.h`, add the following within the curly braces in the `@interface MyDocument` block:

```
@private
MySettings *mySettings;
```

2. To prevent the compiler from complaining about an unknown MySettings type, add the following above `@interface MyDocument`:

```
@class MySettings;
```

3. Add the following method declaration after `@interface MyDocument`:

```
// Accessor methods and conveniences

- (MySettings *)mySettings;
```

   You do not need a `setMySettings:` method, because a MySettings object will be instantiated only once, in MyDocument's initialization method, and you don't want to invite anybody else to instantiate a MySettings object.

4. Still in the header file `MyDocument.h`, you may declare an `init` method after `@interface MyDocument`—but only do this if you feel a need to declare override methods that are commonly overridden, anyway:

```
- (id)init;
```

5. In the source file `MyDocument.m`, add the following definition to initialize the document object by setting its `mySettings` instance variable:

```
// Initialization

- (id)init {
    if (self = [super init]) {
        mySettings = [[MySettings allocWithZone:[self
zone]] initWithDocument:self];
    }
    return self;
}
```

   This method again utilizes the standard Objective-C technique that you first saw in *Step 3.3* for

initializing a subclass of another class. Notice again the tricky test in the first line.

Its allocation of the MySettings object utilizes the standard Objective-C technique, which you have also seen before, for allocating memory for a new object in the application's zone and initializing it, all in one line. As noted in *Step 3.5.2*, above, this invokes the settings object's designated initializer to pass in a reference to the document.

6. Immediately following, add the following `dealloc` method override in `MyDocument.m`:

```
- (void)dealloc {
    [[self mySettings] release];
    [super dealloc];
}
```

It is a general rule of Cocoa programming that any object you allocate by calling `alloc` or `allocWithZone:` must be explicitly released by you when it is no longer needed. The nuances of this important and complex subject will be left for later. In the meantime, you have just made sure that your document releases its associated settings object when the document itself is deallocated. Because only one settings object is created by each document, this takes care of the issue.

You encountered this issue previously, in *Step 3.3*, when you arranged for the document to release its window controller immediately after creating it. There, however, the window controller continued to live because it was added to the document's array of window controllers, which would eventually take care of releasing it for you when the document is closed. Here, your MySettings object is a custom object entirely under your control, and you therefore release it only when the document itself is being deallocated.

7. In `MyDocument.m`, add the implementation of the mySettings accessor method:

```
// Accessor methods and conveniences

- (MySettings *)mySettings {
    return mySettings;
}
```

8. To prevent the linker from complaining when it finds invocations of methods from MySettings, add the following after `#import "MyDocument.h"`:

```
#import "MySettings.h"
```

3.5.4 Define a data variable in the model object

Recall that Interface Builder has already supplied the declaration and definition of the `myCheckboxAction:` method in MyWindowController, complete with the `IBAction` type specification so that it will be recognized as an action method if you later read the header file back into Interface Builder. You will fill in the missing contents of the action method's definition shortly. In order to do that, you must first implement a variable to hold the data, as well as accessors to set it and get it.

1. In the header file `MySettings.h`, declare a variable to hold the value represented by the checkbox control. Insert the following, within the `@interface MySettings` block (that is, between the curly braces), after the declaration of the myDocument instance variable:

   ```
   BOOL myCheckboxValue;
   ```

   In a real application, this variable would be named to describe the item whose on/off or true/false state it records, such as `speechEnabled` or `isEnrolled`.

2. Declare accessor methods and conveniences in the header file `MySettings.h` to set and read the value of the variable. Add these lines at the end of the Accessor methods and conveniences section:

   ```
   - (void)setMyCheckboxValue:(BOOL)value;
   - (BOOL)myCheckboxValue;
   - (void)toggleMyCheckboxValue;
   ```

3. Switch to the source file `MySettings.m` and implement the accessor methods by adding these lines at the end of the Accessor methods and conveniences section:

   ```
   - (void)setMyCheckboxValue:(BOOL)value {
       myCheckboxValue = value;
   }

   - (BOOL)myCheckboxValue {
       return myCheckboxValue;
   }

   - (void)toggleMyCheckboxValue {
       [self setMyCheckboxValue:([self myCheckboxValue] ?
   NO : YES)];
   }
   ```

The last of these, `toggleMyCheckboxValue`, is a red herring. It illustrates how a programmer might initially think to implement a convenience method using the C ternary operator to set the value of `myCheckboxValue` to the reverse of its current value. In a moment, in *Step 3.6*, you will have second thoughts and eliminate this ill-considered method.

The `toggleMyCheckboxValue` method could have been written to obtain the value of the variable directly for simplicity, as follows: `myCheckboxValue = (myCheckboxValue ? NO : YES);`. Although different programmers might make different decisions on this point, Vermont Recipes will not use this shorter technique, accessing variables instead through accessor methods. As you will see later, this decision will make it much easier to implement undo and redo and to add AppleScript functionality, and Cocoa applications therefore generally use accessor methods. (The parentheses around the C ternary operator are not required by C or Objective-C syntax. They are included here, as elsewhere in Vermont Recipes in a variety of contexts, only because it makes the code easier to read.)

Now, any objects that may need to obtain the data value associated with the checkbox user control can invoke the MySettings accessor method `myCheckboxValue`, and they can set the value by invoking `setMyCheckboxValue:`.

3.5.5 Link the window controller to the model object

There is still something missing. The `setMyCheckboxValue:` and `myCheckboxValue` methods are located in the MySettings class where the instance variable holding the data is located, but the action method that will invoke this method to set the data value is located in the MyWindowController object. These two objects do not yet know how to talk to one another. You must supply this final missing link now.

1. The first step is to link MyWindowController to MySettings. This will be done indirectly, through the document object acting as an intermediary. This illustrates what was meant when we said earlier that the document object is a controller of the model object that holds the data.

    a. In the header file `MyWindowController.h`, add the following method declaration at the top of the Accessor methods and conveniences section:

    ```
    - (MySettings *)mySettings;
    ```

    b. Also add the following above `@interface MyWindowController`, so the compiler will recognize that you meant to use the `MySettings` type for the return value of the `mySettings` method:

    ```
    @class MySettings;
    ```

c. In the source file `MyWindowController.m`, define the new mySettings accessor method by adding the following at the top of the Accessor methods and conveniences section:

```
- (MySettings *)mySettings {
    return [[self document] mySettings];
}
```

Notice that this accessor method does not return an instance variable that was declared in the same object, as you have seen done until now. Instead, it simply invokes its associated document's `mySettings` accessor method, completing a chain of links from MyWindowController through MyDocument to MySettings. You learned in *Step 2.6.2.1* that Cocoa's NSDocument class already declares an accessor method, `document`, which is how the first link of this chain was established. Declaring accessor methods as chained links in this fashion is very common in Cocoa.

There are at least two other ways in which you could have given MyWindowController access to the MySettings object.

For one, you could have declared a `mySettings` instance variable in MyWindowController, as you did in MyDocument, and set it to the value of the document's link to MySettings. However, this would require setting aside additional memory for the instance variable, which might grow out of hand if you were to follow this practice routinely in a large application. In general, it is recommended that intermediate instance variables like this be avoided whenever possible.

As a second alternative, you could have dispensed with the accessor method and simply used chained references through the document object every time MyWindowController needs access to a MySettings value. However, the window controller will eventually acquire jursisdiction over a lot of additional user controls, and you would have to invoke something like `[[[self document] mySettings] myCheckboxValue]` for each of them. The shorter form of reference allowed by the accessor method in MyWindowController, `[[self mySettings] myCheckboxValue]`, is a more convenient shortcut. It will also make it easier to change the relationship between the document object and the settings object, if that should prove necessary in the future, by changing only one statement in MyWindowController.m instead of changing code throughout the file.

2. In `MyWindowController.m`, add the following line after `#import MyWindowController.h`:

```
#import "MySettings.h"
```

Without this, the compiler will complain that MyWindowController doesn't understand the `mySettings` command.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 1: A simple, multi-document, multi-window application

Step 4: Provide for data storage and retrieval

4.2 Implement the stub methods for data representation

`MyDocument.m` includes two stub methods provided by the Cocoa Document-based Application template relating to storing data and reading it back from storage. One, `dataRepresentationOfType:`, converts your application's data from its live, internal format or representation into an NSData object suitable for storage. It returns the NSData object, which is simply a byte stream (a string of characters), and Cocoa then stores it for you. The second, `loadDataRepresentation:ofType:`, reads data from storage as a byte stream and converts it back to your document's internal storage representation in RAM. You must implement both methods by providing statements that accomplish the data conversion based on the application's data structures.

You will implement these methods in such a way that a document is saved in property list format. Any application that knows how to parse property list files, such as Apple's PropertyListEditor, will be able to read the file.

The methods you will write here are suitable for storing and retrieving individual items of data represented as C data types, such as the Boolean variable `myCheckboxValue`. In a later Recipe, you will learn about other ways to store and retrieve data, for example, in formal XML format in *Recipe 4, Step 1*, and to tell objects such as MySettings to archive and unarchive themselves as objects.

Be forewarned that the code in this Step is dense. It merits close study because, among other things, it is your first exposure to the Cocoa dictionary class, which is used frequently throughout Cocoa. It implements the key-value technology to which you have already been exposed.

Launch Project Builder and open the project, if necessary.

4.2.1 Convert the document's internal data to its external storage representation

1. In the header file `MyDocument.h`, add declarations before `@end` for two subroutines, as follows:

```
// Persistent storage

    // Saving information to persistent storage

- (NSData *)convertForStorage;
- (NSDictionary *)setupDictionaryFromMemory;
```

2. Switch to the source file `MyDocument.m` and replace the stub method `dataRepresentationOfType:` with the following definition:

```
// Persistent storage

    // Saving information to persistent storage

- (NSData *)dataRepresentationOfType:(NSString *)type {
    if ([type isEqualToString:myDocumentType]) {
        return [self convertForStorage];
    } else {
        return nil;
    }
}
```

   You will define the m`yDocumentType` variable shortly.

3. In `MyDocument.m`, define the two methods whose interface you declared in instruction 2., above, immediately following `dataRepresentationOfType:`.

```
- (NSData *)convertForStorage {
    NSDictionary *dictionary = [self
setupDictionaryFromMemory];
    NSString *string = [dictionary description];
    return [string
dataUsingEncoding:NSUTF8StringEncoding];
}

- (NSDictionary *)setupDictionaryFromMemory {
    NSMutableDictionary *dictionary =
[NSMutableDictionary dictionary];
    [dictionary setObject:NSStringFromClass([self
class]) forKey:myDocumentClassKey];
    [dictionary setObject:[NSString
stringWithFormat:@"%d", currentMyDocumentVersion]
forKey:myDocumentVersionKey];
    [[self mySettings] convertToDictionary:dictionary];
    return dictionary;
}
```

These routines also contain some variables you have not yet defined.

There is a lot going on here. You are urged, as always, to read the documentation for each of the Cocoa methods invoked in these subroutines. The key point is that the `setupDictionaryFromMemory` method uses key-value pairs to set up a Cocoa-standard temporary mutable dictionary representing the document object's data in memory, as an intermediate representation before streaming it to persistent storage.

A note about dictionaries

Temporary "dictionary" objects are used frequently in the Cocoa frameworks and in Cocoa applications as a fast, efficient, and standardized way to encapsulate data and make it available to the application.

A dictionary is a collection of key-value pairs. The keys are usually strings that label the corresponding values, and they are often kept in variables in a running application, for easy use. In the dictionary, the keys are organized into a hash table for fast lookup of the matched values. Some aspects of this key-value pair technique are based on Apple-patented technology.

A dictionary's keys are arbitrary values, unique within any one dictionary. Unlike an array's indices, a dictionary's keys are constant; they do not change value as

entries are added or removed.

The values in a dictionary are unordered. Unlike a set's members, a dictionary's values are always associated with a matching key.

A number of convenient methods are provided in NSDictionary and NSMutableDictionary for managing dictionaries, freeing you from the tribulations of hash tables. The most frequently-used methods may be those for adding an entry to a dictionary, `setObject:forKey:`, and for retrieving a value from a dictionary, `objectForKey:`. Another often-used method is `description`, which returns the entire dictionary as a string formatted as a property list. NSDictionary, like many collection classes in Cocoa, also implements methods to write its contents to persistent storage and read them back.

The `setupDictionaryFromMemory` method returns the dictionary, which is used in turn by the `convertForStorage` method to generate an NSString object that is immediately converted to an NSData object—that is, a byte stream—suitable for writing to disk. In the previous version of the Vermont Recipes application, the conversion to an NSData object was accomplished using ASCII encoding; in the present version, it is done using UTF8 encoding to match Cocoa's recent change to that encoding for property lists. The application's override of the NSDocument `dataRepresentationOfType:` method then returns this byte stream to Cocoa, which writes it to persistent storage for you.

In `setupDictionaryFromMemory`, you invoke NSDictionary's `setObject:forKey:` method, a workhorse that you will use frequently in Cocoa programming. Here, the first two invocations add strings to the dictionary that identify the class of the object whose data is to be written to storage and an integer defining the version of the document format being used. The latter will come in handy if you revise the format and wish to provide for backwards compatibility.

Finally, you pass the dictionary to a `convertToDictionary:` method yet to be written in MySettings, where another value will be added to the dictionary before returning it to you here. You will write that method in *Step 4.2.3*, below, where you will invoke `setObject:forKey:` for a third time. That method will initially represent the value of the `myCheckboxValue` variable as a dictionary entry and pass the dictionary back to the document here. The important point, however, is that later, when you add additional variables to the MySettings object, you will simply add them to the dictionary that MySettings returns to the document object in its `convertToDictionary:` method, without having to make any changes to the document object. In other words, by setting up the document's storage routines in this fashion, you have isolated the internals of the MySettings data representation so successfully that the document object needn't know about them.

4. Define the new variables referenced in instructions 2. and 3., above, by inserting the following in `MyDocument.m` at the top of the Persistent Storage section:

```
// Keys and values for dictionary

NSString *myDocumentType = @"Vermont Recipes 1 Document
Format";
static NSString *myDocumentClassKey = @"Class";
static NSString *myDocumentVersionKey = @"Version";
static int currentMyDocumentVersion = 1;
```

   The `myDocumentType` variable must be assigned the same string that you entered as the `CFBundleTypeName` in the first element of the `CFBundleDocumentTypes` array in the target's Application Settings in *Step 3.1.1*.

4.2.2 Convert the document's stored data to its live internal representation

1. In the header file `MyDocument.h`, add declarations before `@end` for two subroutines, as follows:

```
// Loading information from persistent storage

- (void)restoreFromStorage:(NSData *)data;
- (NSDictionary *)setupDictionaryFromStorage:(NSData
*)data;
```

2. Switch to the source file `MyDocument.m` and replace the stub method `loadDataRepresentation:ofType:` with the following:

```
// Loading information from persistent storage

- (BOOL)loadDataRepresentation:(NSData *)data
ofType:(NSString *)type {
    if ([type isEqualToString:myDocumentType]) {
        [self restoreFromStorage:data];
        return YES;
    } else {
        return NO;
    }
}
```

3. In `MyDocument.m`, add the two methods whose declarations you added in instruction 2, above,

immediately following:

```
- (void)restoreFromStorage:(NSData *)data {
    NSDictionary *dictionary = [self
setupDictionaryFromStorage:data];
    [[self mySettings]
restoreFromDictionary:dictionary];
}

- (NSDictionary *)setupDictionaryFromStorage:(NSData
*)data {
    NSString *string = [[NSString allocWithZone:[self
zone]] initWithData:data
encoding:NSUTF8StringEncoding];
    NSDictionary *dictionary = [string propertyList];
    [string release];
    return dictionary;
}
```

The `restoreFromStorage:` method invokes the MySettings object's `restoreFromDictionary:` method, which you will write shortly. The MySettings `restoreFromDictionary:` method plays a similar role to that played by the `convertToDictionary:` method invoked in `setupDictionaryFromMemory` in *Step 4.2.1*, above. It allows the MySettings object to take a dictionary, which was just read from persistent memory as a byte stream and converted to a dictionary by `setupDictionaryFromStorage:`, and convert it into the form required by the data variables in MySettings. Again, the document's data storage routines do not need to know anything about the data format implemented in MySettings.

4.2.3 Implement the model object's data conversion methods

1. In the `MySettings.h` header file, insert the following declarations:

```
// Persistent storage

    // Saving information to persistent storage:

- (void)convertToDictionary:(NSMutableDictionary
*)dictionary;

    // Loading information from persistent storage:

- (void)restoreFromDictionary:(NSDictionary
*)dictionary;
```

2. In the `MySettings.m` source file, insert the following definitions:

```
// Persistent storage

    // Keys and values for dictionary

static NSString *myCheckboxValueKey =
@"MyCheckboxValue";

    // Saving information to persistent storage:

- (void)convertToDictionary:(NSMutableDictionary
*)dictionary {
    [dictionary setObject:[NSString
stringWithFormat:@"%d", myCheckboxValue]
forKey:myCheckboxValueKey];
}

// Loading information from persistent storage:

- (void)restoreFromDictionary:(NSDictionary
*)dictionary {
    myCheckboxValue = (BOOL)[[dictionary
objectForKey:myCheckboxValueKey] intValue];
}
```

The Boolean values for the checkbox setting are converted to strings in `convertToDictionary:`, since they will be stored as a byte stream. This is accomplished here by using another workhorse Cocoa method, `stringWithFormat:`. It uses standard C printf codes to convert values, here taking the Boolean value in the `myCheckboxValue` variable and converting it to a string representation of an integer. In the `restoreFromDictionary:` method, the reverse is done, taking the integer value of

the string and casting it to a Boolean value. In *Recipe 2* and following, you will add many similar statements to this method to convert other data values to string format for storage. Note that any data that is already in string format does not have to be converted using the `stringWithFormat:` method but can instead be installed directly into the dictionary. Note also that it is not necessary to populate a dictionary with string values; you can also use values consisting of Cocoa objects. You use strings here because of the ease with which it allows you to stream the dictionary to disk as an NSData object and because, later, in *Recipe 4, Step 1*, it will make it easy to write an AppleScript to read the file.

Notice that the `myCheckboxValue` variable is set directly in `restoreFromDictionary:`, rather than by calling its accessor method, `setMyCheckboxValue:`. This is consistent with the temporary device you employed in *Step 4.1*, using direct access to the variable for initialization, because reading data from disk is a form of initialization whether it happens because the user chose File > Open… or File > Revert. As an initialization operation, it should not be registered with the document's undo manager. As you will learn in *Step 5.1*, the accessor method will register changes with the undo manager, so that method is not used here in `restoreFromDictionary:`. Still later, in *Recipe 2, Step 2*, you will use the `setMyCheckboxValue:` accessor method here, after all, employing a better technique to avoid registering with the undo manager.

These two methods belong in the MySettings object, a model object, because they involve direct manipulation of the object's internal data structures. The outside world, including the document object, should know nothing of these internal data structures. All the document needs to know is that they return or accept a dictionary, which the document can store or retrieve without knowing its contents. As you add more data variables to MySettings, you will simply add lines to these two methods to convert the individual variables to and from the dictionary object, without having to make any changes to the document object.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 5: Implement Undo and Redo

In *Step 4*, you wrote methods to convert the document's internal data to a representation suitable for storage. However, the application does not yet have a means to know when the document needs to be saved, because it cannot tell when the document has been altered. You need to add routines to enable the Revert menu item when a document has been modified, and also to cause an alert to be shown if the user attempts to close a document that has been modified.

In Cocoa, you can keep track of document changes explicitly, but it will be accomplished for you automatically if you implement the Cocoa AppKit's built-in undo and redo capability. The undo manager necessarily tracks changes made by the user, and Cocoa therefore uses this information to implement the proper menu and closing behaviors for you. Undo and redo support should be part of every Macintosh application, so you will now take advantage of this labor-saving feature by implementing it and, as a side effect, you will be able to dispense with tracking document changes yourself.

The basic operating principle behind Cocoa's undo and redo support is that changes to the application's state initiated by the user, especially changes to a document's data, should be registered or recorded with the appropriate undo manager object. The undo manager can automatically revert or restore the change later, when the user chooses Undo or Redo from the Edit menu, by using information provided at the time of registration or recording.

It normally makes sense to register a change to a document's data in a so-called "primitive" method which, like `setMyCheckboxValue:` in MySettings, performs the operation by directly altering an instance variable. Every other method that invokes the primitive method will thereby gain the benefit of Cocoa's undo and redo support. As noted previously, every operation that changes the value of the variable should

do so through this primitive method, except that initialization operations should not be undoable and therefore require special attention.

So-called "extended" methods, which invoke a primitive method to effect a change indirectly, should not register with the undo manager because the primitive method will do it for them. An example of an extended method is `toggleMyCheckboxValue`, from *Step 3.5*, which invoked `setMyCheckboxValue:` to do its work and therefore would not have needed to register with the undo manager itself.

Also, recall from *Step 4.1* that initialization of an instance variable should not register with the undo manager. You can safely implement undo registration in a primitive method because, in Recipe 1, you always initialize variables directly instead of calling the primitive method. An example of this is opening a document or reverting to its saved state, which are forms of initialization. For this reason, the `loadDataRepresentation:ofType:` method that you wrote in *Step 4.2.2* did not call the primitive accessor method but instead set the variable directly. Later, in *Recipe 2, Step 2*, you will use another technique to avoid registration with the undo manager, and you will then be able to use the primitive accessor method to set data variables even during initialization.

At this point, the only action you need to deal with for undo and redo is the user's clicking the checkbox to change the `myCheckboxValue` variable. This change is implemented by the `setMyCheckboxValue:` method in MySettings. Therefore, you should implement undo and redo in that method.

After you have implemented undo and redo for the document's data, you will attend to the wording of the Undo and Redo menu items and consider how the application should update its user interface to remain synchronized with the changed data.

5.1 Register data changes with the document's undo manager

Launch Project Builder and open the project, if necessary.

1. In the source file `MySettings.m`, change the `setMyCheckboxValue:` primitive accessor method to the following:

```
- (void)setMyCheckboxValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setMyCheckboxValue:myCheckboxValue];
    myCheckboxValue = value;
}
```

   The invocation of the undo manager's `prepareWithInvocationTarget:` method causes the

document to create an undo manager object "lazily," if one does not already exist. The statement records, or saves in memory, the `setMyCheckboxValue:` method, as if it were being called now using the current state of the `myCheckboxValue` variable; that is, its state before the user clicked on the control. This recorded version of the call will automatically be played back from the undo manager's undo stack when the user chooses Undo from the Edit menu, in order to restore the variable to this recorded value. At the same time, it will record a new undo action, with the old value, and place it on the undo manager's redo stack. Cocoa will play this back from the redo stack when the user chooses Redo.

2. MySettings does not have a way to talk to the undo manager, so you must add one. The undo manager will be associated with the document object, so you will create a chained accessor method just as you did in *Step 3.5.5* to enable the window manager to access the MySettings object. In the header file `MySettings.h`, add this declaration in the Accessor methods and conveniences section after the `myDocument` accessor method:

```
- (NSUndoManager *)undoManager;
```

   In the source file `MySettings.m`, add this definition in the Accessor methods and conveniences section after the `myDocument` accessor method:

```
- (NSUndoManager *)undoManager {
    return [[self myDocument] undoManager];
}
```

3. The application must know how to locate the relevant undo manager stack when the user chooses the Undo or Redo menu item. Cocoa handles this by walking the responder chain from the current first responder. It is possible that it will encounter an undo manager object before reaching the top; for example, one might have been implemented in a custom view object that handles undo and redo commands itself. If it reaches the window object at the top of the chain without encountering an undo manager object, it asks the window object's delegate for a reference to an undo manager object. If a delegate exists that implements the `windowWillReturnUndoManager:` delegate method, Cocoa calls it. If none is found, the window creates its own undo manager and uses that. Here, the name of this method indicates that the window manager is about to return an undo manager to the application (i.e., it "will" do so). By implementing the delegate method in your window controller, you alter Cocoa's default behavior by causing the window to return an undo manager of your choosing instead of returning the undo manager it would otherwise create itself.

A note about delegation

Cocoa makes heavy use of delegation as a means for one object to carry out actions or respond to events on behalf of another object. This is an important factor in the power and flexibility of Cocoa, since it allows a simple Cocoa application to behave appropriately, while at the same time allowing Cocoa developers to create more complex applications by altering chosen aspects of default Cocoa behavior. It is often easier to implement a delegate method in a delegate class than it is to subclass a Cocoa class.

An object that can hand off some of its responsibilities to a delegate always includes a standardized mechanism to appoint a delegate, the `setDelegate:` method. This sets up an instance variable that gives the delegator the ability to talk to the delegate. Typically, a developer of a new class uses Interface Builder to connect the delegate to the delegator's delegate outlet.

The developer of a class that can appoint a delegate tries to anticipate all events that could usefully be handled by a delegate and declares for each such event a so-called "delegate method." When a relevant event occurs at run time, the delegating object first checks to see whether a delegate has been appointed and, if so, whether the delegate implements the delegate method; if so, it calls the delegate method as implemented by the delegate.

The developer of a class that might want to react to an event considers whether an object exists that calls appropriate delegate methods in connection with the event. If so, the developer implements the delegate method in the delegate class. The delegate method will be called automatically by the delegator when the triggering event occurs.

Many delegate methods in Cocoa use "will," "should," or "did" in their names. By convention, a "will" delegate method is invoked by the delegator when the event is about to be handled; the delegate's implementation of the delegate method can cause the delegator to do something beforehand. A "should" delegate method is also invoked by the delegator before the event is handled; the delegate's implementation can, however, prevent anything from happening in response to the event as well as altering what would normally happen. A "did" delegate method is invoked by the delegator just after the event has been handled.

You, too, can write classes that employ delegates. Designing appropriate delegate methods may seem like a black art, but it simply requires a good class structure design and an understanding of how delegates could make use of it. Implementation is trivial; just check that a delegate has been appointed and has implemented a particular delegate method before calling it.

Documentation

To understand what delegation is all about, read Delegating Authority - Cocoa Delegation and Notification, a 1999 article on the Stepwise site by Erik Buck.

Recall that MyWindowController was designated as the window's delegate in Interface Builder, in *Step 2.6.3*. In document-based applications, the document is supposed to handle all of its undo chores, and the window controller therefore should return the document's undo manager. To ensure that the application finds the document's undo or redo stack at this point, add the following delegate method to the source file `myWindowController.m`, after the Accessor methods and conveniences section.

```
// Undo management

- (NSUndoManager
*)windowWillReturnUndoManager:(NSWindow *)window {
    return [[self document] undoManager];
}
```

It is not necessary to declare this method in the header file `MyWindowController.h`, because it is a delegate method.

4. You should clear the undo and redo stacks for the document when it is saved because, in the Vermont Recipes application, saving a document is regarded as committing all changes. You could write an application to allow undo beyond the last save, as is done, for example, in Project Builder. But that is a complication you don't need, here. Add the following line just before `return [self convertForStorage];` in the `dataRepresentationOfType:` method in the source file MyDocument.m:

```
[[self undoManager] removeAllActions];
```

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 1: A simple, multi-document, multi-window application

Step 5: Implement Undo and Redo

5.3 Update the user interface

Finally, undoing and redoing changes to the application's data is all well and good, but the user interface needs to be updated, as well. In this Step you will devise a means by which the MySettings model object can let the window controller know that the user interface needs to be updated to reflect the new state of the data.

The Sketch example application supplied by Apple deals with this problem by brute force. If applied here, that technique would require every primitive method that changes a data value, such as `SetMyCheckboxValue:` in MySettings, to invoke a custom method, called something like `invalidateData`. This method in turn would invoke a custom window controller method, called something like `invalidateUI`, to instruct all of the window's user controls to change state to match the new state of the document's data.

There is a serious problem with this brute force solution, if it were applied here. It would require the window controller to update every user control in the window, even though the user had only undone or redone a change to a single data value. While this is not a problem for a window containing only one or a few user controls, it could introduce a noticeable delay if the window contained many complex controls. Also, it would result in a user control being updated twice when clicked; once, when the user clicks the control, then again, when the `invalidateData` method is invoked and in turn invokes `invalidateUI` to update the user control.

There are at least two available techniques to solve these problems. One is to have the window controller's

action method pass to the model object's `SetMyCheckboxValue:` method the selector for a specific method to update the visible state of the `myCheckbox` user control. The `SetMyCheckboxValue:` method would then invoke that selector using the NSObject protocol's `performSelector:` method to update the user control. Although the window controller's action method already knows how to update its user control, the point is that this process would also be registered with the undo manager, to be played back when the user chooses Undo or Redo.

Here, however, you will use another important Cocoa technique, notifications, instead. Every primitive document method that changes a data value, such as `SetMyCheckboxValue:`, will post a notification describing the change to the default notification center. The notification center will in turn immediately broadcast that notification to any other object that has registered to receive it. The MyWindowController object will register to receive such a notification for every user control it manages. When it receives such a notification, it will invoke a method to update the specific user control whose associated data value triggered the notification. This process will also be registered with the undo manager, so the same notification will be broadcast again when the user chooses Undo or Redo. As a result, the model object will not need an outlet to the window controller object, and when the window object receives a notification because Undo or Redo was chosen, the one, and only the one, user control will be updated. As a bonus, any other object in the application can register to receive notification when a specific data item changes, and take action accordingly, without requiring any further changes to the model object's code.

A note about notifications

The Cocoa AppKit creates by default a notification center, which an application object can use to broadcast notifications to any other objects that register to receive 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 broadcast 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.

There is some processing overhead associated with notifications, but in general they are very efficient and can be used in most situations without concern for performance.

Documentation

To understand what notification is all about, read Delegating Authority - Cocoa Delegation and Notification, a 1999 article on the Stepwise site by Erik Buck.

In either case, you will solve the problem of redundant updating of a user control when a user clicks it by the simple expedient of testing its state first. If its state already corresponds to its associated data value, the window controller will not let it update. This will save whatever time would have been required to redraw the control on the screen.

1. Notifications are recognized by name or by the object from which they originated, or by both. It is customary to assign the name to an external variable in the originating class and to use the variable in every class where it is needed. The variable name should be given a unique prefix to avoid contamination of the global namespace; here, we use "VR" for "Vermont Recipes". In the header file `MySettings.h`, after `@end`, declare the following external variable:

```
extern NSString *VRMyCheckboxValueChangedNotification;
```

2. In the source file `MySettings.m`, define the variable as follows, before `@implementation MySettings`:

```
NSString *VRMyCheckboxValueChangedNotification =
@"MyCheckboxValue Changed Notification";
```

3. In the source file `MySettings.m`, insert the following at the end of the `setMyCheckboxValue:` accessor method:

```
[[NSNotificationCenter defaultCenter]
postNotificationName:VRMyCheckboxValueChangedNotification
object:self];
```

4. In the header file `myWindowController.h`, declare these two methods before the Action methods section:

```
// Interface management

    // Generic view updaters

- (void)updateCheckbox:(NSButton *)control
setting:(BOOL)value;

    // Specific view updaters (from notifications)

- (void)updateMyCheckbox:(NSNotification
*)notification;
```

5. In the source file `MyWindowController.m`, define these two methods as follows before the Action methods section:

```
// Interface management

    // Generic view updaters

- (void)updateCheckbox:(NSButton *)control
setting:(BOOL)value {
    if (value != [control state]) {
        [control setState:(value ? NSOnState :
NSOffState)];
    }
}

    // Specific view updaters (from notifications)

- (void)updateMyCheckbox:(NSNotification *)notification
{
    [self updateCheckbox:[self myCheckbox]
setting:[[self mySettings] myCheckboxValue]];
}
```

The `updateMyCheckbox:` method is specific to this one checkbox; it will be invoked by the

notification that you are about to register with the default notification center. The `updateCheckbox:setting:` method is generic; it may eventually be called by many specific checkbox controls, passing in the control and its underlying data setting for each distinct control, as is done in `updateMyCheckbox:`. It will also be called in instruction 7, below, when the window first loads for a new or opened document, and in *Step 7*, when the user reverts the document to its saved state.

Note that the specific `updateMyCheckbox:` method fetched the setting parameter from the model object. It would have been possible to include this information in the notification itself, freeing the window controller from the necessity of looking up the setting in the document object. Including information in a notification is in many circumstances preferred, in order to maintain a full separation between the object originating the notification and an object receiving it. Here, however, it is the window controller's primary function to talk to the model object, so there is no harm in looking the value up. This does, however, limit the ability of other classes to use the notification; they must know about the model object, as MyWindowController does.

Don't be concerned about the apparent mixing of integer, Boolean and enumeration data types in the `updateCheckbox:setting:` method. It is common in Cocoa programming to treat Boolean values as integers whenever this promotes convenience, and the same goes for enumeration types, which are, of course, implemented as integers. The NSButton class reference document is explicit about this in the case of the `setState:` method, saying "Although using the enumerated constants is preferred, value can also be an integer. If the cell has two states, zero is treated as NSOffState, and a non-zero value is treated as NSOnState. If the cell has three states, zero is treated as NSOffState; a negative value, as NSMixedState; and a positive value, as NSOnState." We cast the `setting` parameter to a Boolean value here only to capture the traditional notion that a two-state checkbox is either on or off, true or false; Cocoa allows a checkbox to assume only these two states by default. You will learn how to implement mixed-state (or three-state) checkboxes in *Recipe 2, Step 3*, instruction 4, where the `setting` parameter will be cast to an integer value to allow the use of three states.

6. All that is left is to register the window controller as an observer of the notification with the default notification center. This is done by inserting the following statement in the `windowDidLoad` override method of the source file `MyWindowController.m`, as shown below.

```
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateMyCheckbox:)
name:VRMyCheckboxValueChangedNotification object:[self
mySettings]];
```

You might have been tempted to register the window controller as an observer in the MyWindowController `init` method, instead, but this would involve a subtle error, common among Cocoa beginners. Since the `[self mySettings]` object passed as the `object`

parameter in the registration method would not yet have been created at the time when the window controller is being initialized, it would have been passed as `nil`. Vermont Recipes would appear to function correctly for the time being. However, it would not have functioned correctly later, when additional controls will be added to the application, and it would have been very difficult to debug the problem. To avoid errors, the window controller must be registered with the notification center in the window controller's `windowDidLoad` method, after the MySettings object has been created and initialized and can therefore be passed to the notification center.

It is important to understand exactly why you should not register observers of data changes in the window controller's `init` method. You want your notifications to go only to the update method in the particular instance of MyWindowController that is associated with this MySettings object and its window, not is some other instance representing a different document. If you were to register your observer in the window controller's `init` method, the MySettings object would not yet have been created and you would therefore have passed `nil` in the `object` parameter. As the NSNotificationCenter documentation notes, this causes the notification to be broadcast too widely. If the notification were broadcast to all instances of MyWindowController, then the same user control in every open window would update, potentially causing pending edits in those windows to be aborted and reverted to their original values behind the user's back. In general, you have to be very careful what you do in your window controller's `init` method, because many objects aren't yet in existence when it is invoked. For example, objects instantiated in the nib file do not yet exist. For this reason, many operations must be deferred to the `awakFromNib` method declared in the NSNibAwaking informal protocol, or to an override of NSDocument's `windowControllerDidLoadNib:` method or, in a document-based application like Vermont Recipes, to an override of NSWindowController's `windowDidLoad` method.

Back in instruction 6 of *Step 3.3*, you removed MyWindowController from the list of notification center observers when MyWindowController was deallocated, without understanding why. Now you know: here you have added MyWindowController to the list of observers, and removing it later is a necessary part of the object's cleanup process.

7. Finally, you will recognize that the statement within the new `updateCheckbox:` method in `MyWindowController.m` is identical in effect to a statement you inserted in `windowDidLoad` in *Step 4.3*. Therefore, since you now have a method that performs this function, you should revise `windowDidLoad` to invoke the new method, for the sake of efficiency and easy maintenance. Replace the second line of `windowDidLoad` with the following:

```
[self updateCheckbox:[self myCheckbox] setting:[[self
mySettings] myCheckboxValue]];
```

The cumulative effect of the code you wrote in this Step is that, whenever an item of data in the application is changed by any operation, including an undo or redo operation, the window controller gets

wind of it and updates the associated user control to match. This is all managed in such a way that the model object does not have to know what user interface objects exist, or even that a window controller exists. You could take this same model object, unchanged, and tack a completely different user interface onto it.

That's all there is to implementing undo and redo support in the application. If you compile, link and test the application now, try clicking several times in succession on the checkbox control. Then try undoing these changes from the Edit menu. You will discover that you can repeatedly undo the changes until the Undo menu item finally dims. This is because Cocoa implements unlimited undo by default; it lets you undo each of your clicks until the control finally reverts to its original state. At that point, you can also choose the Redo menu item repeatedly until all of the undos have been reversed and the Redo menu item finally dims.

Best of all, by implementing undo and redo, the application automatically gained the ability to know when a document has been modified and needs to be saved. You can test this by clicking the checkbox in the window to change its state, then closing the window. A sheet will open, advising you that the document has been modified and asking you whether to save it. If you cancel and choose Edit > Undo, you will be able to close the document without seeing this sheet, because the document will know that the document is no longer modified from its previously-saved state.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 3: Checkboxes (switch buttons) in a borderless group box

- Highlights:
  - Using a mixed-state checkbox
  - Using an Objective-C category to enhance a built-in Cocoa class

The checkbox you implemented in *Recipe 1* is sometimes known as a two-state checkbox or switch button. It can be either on (checked) or off (unchecked, or empty). You are also able to create mixed-state checkboxes. They have the familiar checked and unchecked states, but they also have an indeterminate or mixed state indicated by a dash inside the checkbox. The Mac OS 8 Human Interface Guidelines (which are inherited by Aqua except as revised in the Aqua Human Interface Guidelines) describe a mixed-state checkbox as follows:

> There is a mixed state for checkboxes, which shows that a selected range of items has some in the on state and some in the off state. For example, a text formatting checkbox for bold text would be in the mixed state if a text selection contained both bold and non-bold text.

The guidelines go on to remind us that

> Checkboxes differ from radio buttons in that they are independent of each other, even when they offer related options. Any number of checkboxes can be on, off, or mixed at the same time.

For purposes of demonstrating how to implement a mixed-state checkbox, you will create three independent but related two-state checkboxes in a titled group box, and the box will contain a fourth, mixed-state checkbox at the bottom that can be used to turn all three of the two-state checkboxes on or off at once. The mixed-state checkbox will assume the mixed state when one or more, but not all, of the other three checkboxes have been turned on separately, but the user cannot create a mixed state by clicking on this checkbox.

The checkboxes in this Recipe will be grouped because all of them relate to a single subject, but the group will not contain a visible border. The Aqua Human Interface Guidelines recommend that bordered group boxes be used sparingly, if at all, expressing a preference for the use of "white space" to separate a group of related controls visibly from other interface items and groups. This differs from the classic interface guidelines, which encouraged use of enclosing borders to group related items. In Mac OS X, the trend is to retreat from the overly busy user interfaces that have become common in classic applications. In this step, you will use Interface Builder to group the checkboxes in a box because this is convenient for the developer, but you will make the border invisible in order to comply with the guidelines. In *Step 4* you will implement a similar group of checkboxes using a visible bordered box, which is still permitted by the guidelines when it serves usability.

Adding a new control and its associated data variable to an application can be a surprisingly tedious affair, as you will discover in this Step. There are a great many details to take care of. However, the object-oriented structure of a Cocoa application allows the process to be organized into a fixed set of steps, no matter what kind of user control is involved. This allows the process to be routinized to reduce the chance of errors. In this way, features like multiple undo and redo and data storage are taken care of easily. Before turning to the details, here is a checklist of what must be done.

- Use Interface Builder to draw the control, then turn to Project Builder to code it
- **User control outlet variable and accessors**. Add an outlet variable and accessor methods to the window controller, so the state of the control can be accessed
- **Data variable and accessors**. Add a data variable and accessor methods to the model object, so the value represented by the control can be accessed; the `set…` accessor method will also register a data change with the undo manager and post a notification to indicate that the data has changed
- **Notification variable**. Declare a notification variable in the model object, so the model object can notify the window controller when the data changes
- **GUI update method**. Add a method to the window controller to update the visible state of the control in response to notification that the data has changed
- **Notification observer**. Register the window controller as a notification observer, so it will receive notification that the data has changed
- **Action method**. Add an action method to the window controller to change the data in the model object when the user clicks the control
- **Localizable.strings**. Update the `Localizable.strings` file with the new undo and redo menu names
- **Initialization**. Initialize the data variable in the model object to a default value, if desired
- **Data storage**. Add keys and revise methods in the model object to save and retrieve the data to and from persistent storage
- **GUI update method invocation**. Add an invocation of the control updater to the window controller's `updateWindow` method
- In Interface Builder, read in the files in order to capture the new outlet and action, then connect them with the control and set any required delegate connections

It took all of Recipe 1 to cover these steps for the Checkbox control, along with the basics of creating an application. Here, you will do it for the new mixed-state checkbox and its associated two-state checkboxes in a single Step. This road map will serve you well in Steps to come, where you will create many more controls.

After working through this roadmap in detail here, the instructions in subsequent Steps can be shortened in order to help you focus on what is unique about each control.

The routine steps needed to implement these controls are covered in instructions 1. and 2., below. The interesting material—the methods to update the mixed-state checkbox and its action method, and a foray into the land of categories—is covered in instructions 3. and following.

1. Use Interface Builder to create several new checkboxes grouped in a box, using the techniques you learned in *Recipe 1, Step 2.2*. When you are done with this instruction 1, the group should look like that shown in *Screenshot 2-2*.

    a. Open `MyDocument.nib` in Interface Builder and select the Buttons tab view item, if necessary. Remember that the Buttons tab view item must have a broad line around it, so that controls dropped into it from the palette land on the tab view item and don't fall through into the underlying tab view. You can verify that the tab view item is selected by checking the Info palette, which should show the NSTabViewItem Info palette, not the NSTabView Info palette.

    b. From the Views palette, drag three Switch controls onto the Buttons pane and line them up vertically some distance beneath the existing Checkbox control. Ignore the Aqua guideline showing the proper distance below the existing Checkbox control, because you are going to group the three new checkboxes separately, but line all three of them to the left guideline as a convenient way to align them vertically. If necessary, you can select all three of the new checkboxes, then align them vertically by using the Layout > Alignment > Align Left Edges menu item.

    c. Rename the three new switches **`Triangle`**, **`Square`**, and **`Round`**, respectively.

    d. From the Views palette, drag another checkbox onto the Buttons pane, then position it a greater distance below the first three switches than the guideline suggests. Rename it **`Select All`**.

    e. From the Views palette, drag the horizontal line below the box icon onto the Buttons pane, placing it between the top three buttons and the bottom button to serve as a divider. Now you can use the horizontal guidelines to move the divider and the checkbox below it to their proper vertical positions.

    f. Select all four new switches and the divider, then choose Layout > Group in Box. A box with selection handles appears, surrounding the selected items by a distance dictated by the Aqua guides. Change the title of the box to **`Pegs for Tots`**: (with a trailing colon because the box will be made invisible for this group).

    You could have created the box by dragging the box icon from the Views palette onto the Buttons pane, then manually repositioning and resizing it to surround the controls. But the Group in Box command is more convenient because it does the positioning and spacing for you, and it allows you to drag the box and its contents as a group without first making sure you have selected all of them. More importantly, it lets you take advantage of the convenient sizing and spacing tools provided by the NSBox Info palette. You can drag the box and its contained items

now to see that Aqua guides suggest positioning the new checkboxes relative to the existing Checkbox control based on the position of the visible border of the NSBox object.

However, this group box is specified to be borderless, so, in the Attributes pane of the NSBox Info palette, click the button designating a borderless group box. If you watch closely, you see that the individual items within the box move apart slightly. Now drag the box and the items within it, and note that the Aqua guides suggest a slightly different position relative to the existing Checkbox control. This results in placing the group box title too close to the Checkbox control, considering that this group is to be separate, so drag it a pleasant distance further below the Checkbox control. The Aqua Human Interface Guidelines give you flexibility in the use of white space to separate groups of related controls.

g. If you were to reposition the items manually, you would discover a variety of potentially useful tools. Experiment as follows:

    i. Select the three topmost checkboxes in the group, then choose the Tools > Alignment… command to open the Alignment palette. In this palette, you can use the top left button to align the left edges again, if necessary. Then use the bottom right button to spread them out or close them up vertically with appropriate spacing—say, 7 pixels for switches, to achieve the recommended 20-pixel spacing between the baselines of their titles. Finally, adjust the vertical position of the divider so that it is, say, 8 pixels below the "Round" switch, and move the Select All switch so that it is, say, 8 pixels below the divider.

    ii. If you don't like the resulting appearance, drag the checkboxes and the divider individually within the group box. You notice that Aqua guides appear, suggesting internal placements for items within the group box.

    iii. Click one of the checkboxes, then hold down the Option key as you drag over an adjacent control. You see arrows showing the exact number of pixels between them.

    iv. Select the group box and use the Layout > Size to Fit command to shrink the box around the controls. You see that all of the controls close up and the border shrinks around them. The result is not pretty. It appears that this is command is designed to shrink the arrangement to the smallest possible size without overlap, rather than to comply with guidelines.

    v. Turn to the NSBox Info palette and see what tools it offers to change the spacing and appearance of the group box. In Mac OS X 10.0, the only choice for the group title is to have a title or not have a title. You can use the Format menu to change the font and style, but the Aqua Human Interface Guidelines do not offer any encouragement to depart from the defaults. You have a choice of three Box Type buttons, but for this Step we have specified borderless.

You have probably by now thoroughly messed up the spacing and position of the group box and its items. The easiest way to restore them is to start with the topmost item in the group box and drag each in turn to the position indicated by the Aqua guides.

Examine the Aqua Human Interface Guidelines book to satisfy yourself that you have complied with its arrangement, sizing and spacing recommendations. One issue to note is that Mac OS X favors a center-biased dialog layout; at the end of this Step you will adjust the horizontal placement of items in the window to comply with this recommendation.

2. Use Project Builder to write the code required to make the three new two-state checkboxes work, and to implement the simpler features of the fourth, mixed-state checkbox. The Triangle, Square and Round checkboxes will be coded exactly as the Checkbox control was coded, each acting independently of the other to set an associated variable in the MySettings model object.

   The mixed-state Select All checkbox will have new functionality, setting or clearing all of the other checkboxes when it is checked or unchecked. It will in turn be checked, cleared or left in the mixed state, as appropriate, when any of the other checkboxes is checked or unchecked. You will defer the more interesting code involving the mixed-state checkbox to instruction 3, below.

   a. **User control outlet variable and accessors**. Each of the four new checkboxes will be represented by an outlet variable in MyWindowController, so that the window controller can tell the controls to change their visible state when the data changes or obtain the state when that information is needed. In *Recipe 1, Step 2.6.2.2*, you were able to create an outlet for the Checkbox control in Interface Builder. You can't do it that way here, because the MyWindowController files already exist; if you tried to use Interface Builder's Create Files… command now, you would overwrite them. You must therefore create the outlets in code, following the model of *Recipe 1, Step 3.4*.

      In the header file `MyWindowController.h`, declare four new outlets after the `myCheckbox` declaration, as follows:

```
// Pegs switch button group
IBOutlet NSButton *trianglePegsCheckbox;
IBOutlet NSButton *squarePegsCheckbox;
IBOutlet NSButton *roundPegsCheckbox;
IBOutlet NSButton *allPegsCheckbox;
```

      Still in `MyWindowController.h`, also declare accessors for the outlets after the `myCheckbox` accessor, as follows:

```
// Pegs switch button group
- (NSButton *)trianglePegsCheckbox;
- (NSButton *)squarePegsCheckbox;
- (NSButton *)roundPegsCheckbox;
- (NSButton *)allPegsCheckbox;
```

      Turn to the source file `MyWindowController.m` and define the accessors after the `myCheckbox` accessor, as follows:

```
// Pegs switch button group

- (NSButton *)trianglePegsCheckbox {
    return trianglePegsCheckbox;
}

- (NSButton *)squarePegsCheckbox {
    return squarePegsCheckbox;
}

- (NSButton *)roundPegsCheckbox {
    return roundPegsCheckbox;
}

- (NSButton *)allPegsCheckbox {
    return allPegsCheckbox;
}
```

Now that you've gone to all that trouble, let it be said that using accessors to get user controls may be overkill. Everybody recommends using accessors for data variables in your model object, but the justifications for doing so don't necessarily extend to user controls. The implementation of user controls, unlike your application's data structures, is pretty much built into Cocoa. It is unlikely to change in ways that you would want to hide from your header files. Your code would be simpler if you just accessed user controls by their instance variables directly. Nevertheless, on the theory that tutorial code should be conservative, you will continue to use accessors for user controls throughout Vermont Recipes. It may even turn out that there is a payoff down the line, if we think about automating the GUI of Vermont Recipes using AppleScript.

b. **Data variable and accessors**. Three of the new checkboxes require corresponding variables in MySettings to hold the data they represent. The fourth checkbox is used only to affect or reflect the state of the other three as a group, so it does not require an independent data variable. You can determine the state of the three as a group by testing all of them at once, and this avoids the risk that a fourth variable to track the state of the group might get out of sync.

   You will now create the three new data variables in MySettings, along with their accessors, following the model of *Recipe 1, Step 3.5.4*. You will include in the `set…` methods the undo manager and notification center statements that were taught in *Recipe 1, Step 5.1* and *Recipe 1, Step 5.3*, in order to ensure that changing the values of these data variables will be undoable and will be reflected in the graphical user interface.

   In the header file `MySettings.h`, declare three new variables after the `myCheckboxValue` variable, as follows:

```
// Pegs
BOOL trianglePegsValue;
BOOL squarePegsValue;
BOOL roundPegsValue;
```

In `MySettings.h`, also declare the corresponding accessor methods after the `myCheckboxValue` accessors, as follows:

```
// Pegs

- (void)setTrianglePegsValue:(BOOL)value;
- (BOOL)trianglePegsValue;

- (void)setSquarePegsValue:(BOOL)value;
- (BOOL)squarePegsValue;

- (void)setRoundPegsValue:(BOOL)value;
- (BOOL)roundPegsValue;
```

Turn to the source file `MySettings.m` and define the accessor methods after the `myCheckboxValue` accessors, as follows:

```
// Pegs

- (void)setTrianglePegsValue:(BOOL)value {
    [[[self undoManager] prepareWithInvocationTarget:self]
setTrianglePegsValue:trianglePegsValue];
    trianglePegsValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRTrianglePegsValueChangedNotification
object:self];
}

- (BOOL)trianglePegsValue {
    return trianglePegsValue;
}

- (void)setSquarePegsValue:(BOOL)value {
    [[[self undoManager] prepareWithInvocationTarget:self]
setSquarePegsValue:squarePegsValue];
    squarePegsValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRSquarePegsValueChangedNotification
object:self];
}
```

```
- (BOOL)squarePegsValue {
    return squarePegsValue;
}

- (void)setRoundPegsValue:(BOOL)value {
    [[[self undoManager] prepareWithInvocationTarget:self]
setRoundPegsValue:roundPegsValue];
    roundPegsValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRRoundPegsValueChangedNotification
object:self];
}

- (BOOL)roundPegsValue {
    return roundPegsValue;
}
```

c. **Notification variable**. Return to the header file `MySettings.h`, at the bottom, to declare the notification variables used in the `set`... methods in order to cause the window controller to update the graphical user interface when the data is changed, as follows:

```
// Pegs
extern NSString
*VRTrianglePegsValueChangedNotification;
extern NSString *VRSquarePegsValueChangedNotification;
extern NSString *VRRoundPegsValueChangedNotification;
```

Turn back to the source file `MySettings.m`, near the top, to define the notification variables, as follows:

```
// Pegs
NSString *VRTrianglePegsValueChangedNotification =
@"TrianglePegsValue Changed Notification";
NSString *VRSquarePegsValueChangedNotification =
@"SquarePegsValue Changed Notification";
NSString *VRRoundPegsValueChangedNotification =
@"RoundPegsValue Changed Notification";
```

d. **GUI update method**. Go now to the header file `MyWindowController.h` to declare methods to update the graphical user interface in response to these notifications, after `updateMyCheckbox:` in the `Specific view updaters` section, as follows:

```
// Pegs
- (void)updateTrianglePegsCheckbox:(NSNotification
*)notification;
- (void)updateSquarePegsCheckbox:(NSNotification
*)notification;
- (void)updateRoundPegsCheckbox:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define these specific update methods, after `updateMyCheckbox:` in the `Specific view updaters` section, as follows:

```
// Pegs

- (void)updateTrianglePegsCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self trianglePegsCheckbox]
setting:[[self mySettings] trianglePegsValue]];
}

- (void)updateSquarePegsCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self squarePegsCheckbox]
setting:[[self mySettings] squarePegsValue]];
}

- (void)updateRoundPegsCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self roundPegsCheckbox]
setting:[[self mySettings] roundPegsValue]];
}
```

It is not necessary to declare a new generic checkbox updater for these three update methods, because they call the generic checkbox updater you created in *Recipe 1, Step 5.3*. That generic method handles two-state checkboxes, and that's what these checkboxes are.

Notice, however, that you have not implemented an update method for the Select All checkbox. You will have to do so, obviously, but you will defer this task for now. The Select All checkbox updater will require more careful thought, because it must reflect the state of all three of the new data variables in combination. Among other things, it must be able to display the dash that characterizes a mixed-state checkbox. To do this, you will have to write a new generic update method to handle mixed-state checkboxes. You will return to this issue later in this Step, in instruction 4.

e. **Notification observer**. Now register the window controller as a notification observer of the

notifications that will trigger these updaters, by inserting the following statements in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the existing registration:

```
// Pegs
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateTrianglePegsCheckbox:)
name:VRTrianglePegsValueChangedNotification
object:[self mySettings]];
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateSquarePegsCheckbox:)
name:VRSquarePegsValueChangedNotification object:[self
mySettings]];
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateRoundPegsCheckbox:)
name:VRRoundPegsValueChangedNotification object:[self
mySettings]];
```

f. **Action method**. Next, you must add action methods that will be triggered when the user checks or unchecks any of the new controls, in order to update the data variables on the model of *Recipe 1, Step 3.6*. Again, because the files already exist, you cannot create stubs for these action methods in Interface Builder, as you did for the `myCheckbox` action method in *Recipe 1, Step 2.6.1*, for fear that the Create Files… command will overwrite your code files. In the header file `MyWindowController.h`, after the existing `myCheckboxAction:` method at the end, add the following:

```
// Pegs
- (IBAction)trianglePegsAction:(id)sender;
- (IBAction)squarePegsAction:(id)sender;
- (IBAction)roundPegsAction:(id)sender;
```

In the source file `MyWindowController.m`, define these action methods, after the existing `myCheckboxAction:` method at the end, as follows:

```
// Pegs

- (IBAction)trianglePegsAction:(id)sender {
    [[self mySettings] setTrianglePegsValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Triangle Pegs",
@"Name of undo/redo menu item after Triangle checkbox
control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Triangle
Pegs", @"Name of undo/redo menu item after Triangle
checkbox control was cleared")];
    }
}

- (IBAction)squarePegsAction:(id)sender {
    [[self mySettings] setSquarePegsValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Square Pegs",
@"Name of undo/redo menu item after Square checkbox
control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Square Pegs",
@"Name of undo/redo menu item after Square checkbox
control was cleared")];
    }
}

- (IBAction)roundPegsAction:(id)sender {
    [[self mySettings] setRoundPegsValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Round Pegs",
@"Name of undo/redo menu item after Round checkbox
control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Round Pegs",
@"Name of undo/redo menu item after Round checkbox
```

```
control was cleared")];
    }
}
```

Notice that you have not implemented an action method for the Select All checkbox. You will defer this, too, to instruction 3. It will require more careful thought, because it must change the state of all three of the new data variables in combination.

g. **Localizable.strings**. When you add user controls you generally also add action methods, and action methods generally name undo and redo menu items. These are strings that must be localizable. So don't forget to update the `Localizable.strings` file by adding all of the localizable strings used here ("Set Triangle Pegs" and so on), along with their comments. Follow the model of the localized strings you added to the file in *Recipe 1, Step 5.2* for the Checkbox control.

h. **Initialization**. In *Recipe 1, Step 4.1*, you initialized the value of the `myCheckboxValue` variable to `YES`. Just to assist in testing, you will now set the value of the `trianglesValue` variable to `YES`, as well. You will not initialize the other new data variables, and Objective-C will therefore initialize them to the default value `NO`. Some programmers prefer to initialize values to `0` or equivalent explicitly, for the sake of clarity. You won't do that here because you can in fact rely on Objective-C to do this for you, and in any event you will implement a full-blown preferences system in a later Recipe.

   In the source file `MySettings.m`, add this line to the `initWithDocument:` method, after the initialization of `myCheckboxValue`:

```
[self setTrianglePegsValue:YES];
```

   There is something new you can do in this connection. Return to Interface Builder, select the Triangle checkbox and, in the Options area of the NSButton Info palette's Attributes pane, check the Selected checkbox. A check will appear in the Triangle switch. Now, when the application launches and a document opens, the Triangle checkbox will appear in its default on state without delay or flicker because the nib file and the default data value are the same. As long as you're in Interface Builder, you might as well do the same with the Checkbox switch from *Recipe 1*, since it is also initialized to `YES`.

i. **Data storage**. You must attend to persistent storage of the new data variables. Thanks to the architecture you have set up for this in MySettings, the task is extremely easy. These are all Boolean values, and your code is essentially identical to that used to save and load the Boolean value of the `myCheckBoxValue` variable in *Recipe 1, Step 4.2.3*.

   At the end of this Step, however, you will return to this code and improve it. To preview the change, notice that the Boolean values for these variables are saved as integers: `1` for `YES` and `0` for `NO`. In the event you ever examine Vermont Recipes documents using a utility that can read

property lists, it may be difficult to distinguish actual integer values in the file (which may have values much higher than 1 or 0) from Booleans masquerading as integers. In instruction 7., below, you will learn a remarkably convenient way to save Booleans as "YES" or "NO" strings, improving the human readability of your documents. But, for the moment, go ahead and do it the old way.

In the source file `MySettings.m`, define these keys in the Keys and values for dictionary subsection of the Persistent storage section:

```
// Pegs
static NSString *trianglePegsValueKey =
@"TrianglePegsValue";
static NSString *squarePegsValueKey =
@"SquarePegsValue";
static NSString *roundPegsValueKey =
@"RoundPegsValue";
```

Immediately after that, add these lines at the end of the `convertToDictionary:` method:

```
// Pegs
[dictionary setObject:[NSString
stringWithFormat:@"%d", trianglePegsValue]
forKey:trianglePegsValueKey];
[dictionary setObject:[NSString
stringWithFormat:@"%d", squarePegsValue]
forKey:squarePegsValueKey];
[dictionary setObject:[NSString
stringWithFormat:@"%d", roundPegsValue]
forKey:roundPegsValueKey];
```

And add these lines near the end of the `restoreFromDictionary:` method. Take care to place them inside the bracketing calls to `[[self undoManager] disableUndoRegistration]` and `[[self undoManager] enableUndoRegistration]`:

```
// Pegs
[self setTrianglePegsValue:[[dictionary
objectForKey:trianglePegsValueKey] intValue]];
[self setSquarePegsValue:[[dictionary
objectForKey:squarePegsValueKey] intValue]];
[self setRoundPegsValue:[[dictionary
objectForKey:roundPegsValueKey] intValue]];
```

j. **GUI update method invocation**. Finally, add calls to the user control update methods to the window controller, so that the controls will be drawn to correctly reflect the data when the document is created or opened. In `MyWindowController.m`, add the following calls at the end of the `updateWindow` method:

```
// Pegs
[self updateTrianglePegsCheckbox:nil];
[self updateSquarePegsCheckbox:nil];
[self updateRoundPegsCheckbox:nil];
```

You will have to come back to these two methods in a moment to update the `allPegsCheckbox` control, too.

3. You have left until the end the most interesting parts of this Step, implementing the method to update a mixed-state checkbox and implementing its action method. You will tackle the action method first, because it is easier.

   In the header file `MyWindowController.h`, declare the action method as follows, at the end of the file:

```
- (IBAction)allPegsAction:(id)sender;
```

   Then, in the source file `MyWindowController.m`, define it at the end of the file, as follows:

```
- (IBAction)allPegsAction:(id)sender {
    int newState;

    if ([sender state] == NSMixedState) {
        [sender setState:NSOnState];
    }
    newState = [sender state];

    [[self mySettings] setTrianglePegsValue:newState];
    [[self mySettings] setSquarePegsValue:newState];
    [[self mySettings] setRoundPegsValue:newState];

    if (newState == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set All Pegs", @"Name of
undo/redo menu item after Select All checkbox control was
set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear All Pegs", @"Name
```

```
of undo/redo menu item after Select All checkbox control
was cleared")];
    }
}
```

There are some subtleties here. When a user clicks a control, Cocoa updates the control's state before invoking its action method. Thus, when the action method is called, it has access to the new state of the control (`sender`). In the case of a mixed-state checkbox, the default progression of states as the user clicks the control repeatedly is from `NSOnState` to `NSOffState` to `NSMixedState` then back to `NSOnState`, and so on. While you want this checkbox to be able to display the mixed state when you click other associated checkboxes so as to leave some on and some off, it makes no sense to allow the user to select the mixed state by clicking on the mixed-state checkbox. Therefore, this action method first reads the new state of the control, then forces it to `NSOnState` if its new state is found to be `NSMixedState`.

The action method then saves the control's new (possibly altered) state in the `newState` local variable, in order to use it to set the new value of all three of the other switches to match. A variable is prudent to preserve the new state, because the actual state of the switch may change in response to the notification which each of the other three checkboxes issues in turn when its state is changed by this action method. The details of this are discussed in instruction 4, below.

When the user clicks the `allPegsCheckbox` control, therefore, the action method tells the model object to set the three associated data variables, `trianglePegsValue`, `squarePegsValue` and `roundPegsValue`, to match the new on or off state of the control, all at once. Each of the three `set`… methods invoked here automatically sends a notification that its associated data value has changed, and the corresponding checkbox view updaters are already registered to receive those notifications. They will therefore update the three controls' visible states automatically. The Undo/Redo menu item will read either "Set All Pegs" or "Clear All Pegs," as appropriate. Note that the last point confirms the wisdom of your decision in *Recipe 1, Step 5.2* to follow Apple's recommendation that you set the name of the Undo/Redo menu item in the action method instead of the `set`… accessor method; if you did this in the `set`… method, instead, the view would not have been updated here.

Before you go on, don't forget to add two new strings from this action method to the `Localizable.strings` file, on the model of those you added in instruction 2.g., above.

4. Now for the update method.

    a. Designing a generic method to update the appearance of a mixed-state control is simple, so you will do that first.

    In the header file `MyWindowController.h`, declare the generic method as follows after `updateCheckbox:`:

```
- (void)updateMixedCheckbox:(NSButton *)control
setting:(int)value;
```

In the source file `MyWindowController.m`, define the generic method as follows after `updateCheckbox:`:

```
- (void)updateMixedCheckbox:(NSButton *)control
setting:(int)value {
    if (value != [control state]) {
        [control setState:value];
    }
}
```

This is so simple because the `value` passed in the `setting` parameter will always be one of `NSOnState`, `NSOffState`, or `NSMixedState` or their integer equivalents (0 for `NSOffState`, positive for `NSOnState`, and negative for `NSMixedState`).

b. The implementation of the specific method to update this particular mixed-state checkbox is more complex. It must get the values of the three associated data variables, then set the control's visible state to one of three values depending on whether all are `YES`, all are `NO`, or some are `YES` and others `NO`. You will obtain the desired state in a separate utility method, because this information has to be gathered in more than one place.

In the header file `MyWindowController.h`, declare the utility method as follows, after `updateRoundPegsCheckbox:` at the end of the Specific view updaters section:

```
- (int)wantAllPegsCheckboxState;
```

In the source file `MyWindowController.m`, define the utility method as follows after `updateRoundPegsCheckbox:` at the end of the `Specific view updaters` section:

```
- (int)wantAllPegsCheckboxState {
    if ([[self mySettings] trianglePegsValue] == YES
&& [[self mySettings] squarePegsValue] == YES &&
[[self mySettings] roundPegsValue] == YES) {
        return NSOnState;
    }
    else if ([[self mySettings] trianglePegsValue] ==
NO && [[self mySettings] squarePegsValue] == NO &&
[[self mySettings] roundPegsValue] == NO) {
        return NSOffState;
    }
```

```
    else {
        return NSMixedState;
    }
}
```

c. Now you are ready to write the specific updater method. In the header file `MyWindowController.h`, declare the specific method as follows after `wantAllPegsCheckboxState`:

```
- (void)updateAllPegsCheckbox:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define the specific method as follows after `wantAllPegsCheckboxState`:

```
- (void)updateAllPegsCheckbox:(NSNotification
*)notification {
    [self updateMixedCheckbox:[self allPegsCheckbox]
setting:[self wantAllPegsCheckboxState]];
}
```

d. You now face a tough question: How does the `updateAllPegsCheckbox:` method get invoked? The answer is simple, when you think it through. The Select All switch must potentially be updated to reflect the state of the three data values as a group whenever any one of them is changed. You have already provided for notifications to be broadcast whenever any one of them is changed. Therefore, a straightforward solution is to have the `updateAllPegsCheckbox:` method respond to all of these notifications. To do this, the `updateAllPegsCheckbox:` method must be registered to receive notifications from any of them. It is permissible to have more than one method in a class receive the same notification. In the source file `MyWindowController.m`, add the following lines to the `registerNotificationObservers` method, following the other notification registrations:

```
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateAllPegsCheckbox:)
name:VRTrianglePegsValueChangedNotification
object:[self mySettings]];
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateAllPegsCheckbox:)
name:VRSquarePegsValueChangedNotification object:[self
mySettings]];
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(updateAllPegsCheckbox:)
name:VRRoundPegsValueChangedNotification object:[self
mySettings]];
```

e. This solution, however, raises a new issue when the user checks or unchecks the mixed-state switch itself. In this case, the `allPegsAction:` action method invokes all three of the data accessor methods in order to change their values. An issue arises because the three accessor methods in turn post three separate notifications that their values have changed, so that the visible states of the three controls will update. Unfortunately, the `updateAllPegsCheckbox:` notification method also executes three times in succession because it, too, is observing these notifications. It doesn't need to be called at all in this case, because the mixed-state switch was updated by Cocoa when the user clicked on it. These three redundant calls do not cause a significant delay in Vermont Recipes, where only three accessor methods are invoked, but a better solution might be required if a larger number of user controls were at issue. You should fix this issue now, so it doesn't come back to bite you later if you revise this part of the interface.

   The simplest solution is to create a state variable in MyWindowController and test it in the `updateAllPegsCheckbox:` notification method. It can be a reusable variable, available to any routine, like the `allPegsAction:` method, that controls the state of other checkboxes in a group. In outline, this is the strategy: Name the state variable `controlUpdatingDisabled`. In the `allPegsAction:` action method, call a `disableControlUpdating:` method at the beginning to temporarily disable processing of the notification, and call its sister method `enableControlUpdating` at the end of the action method to reenable processing of the notification. The notification will still be broadcast so that the other three checkboxes in the group can update, but when the notification is received by the `updateAllPegsCheckbox:` notification method, nothing will happen.

   Some programmers will be quite uncomfortable with the use of a state variable in the window controller for this purpose, but there is no obvious way to avoid it here. The notifications must be posted by the data accessor methods so that their associated controls will be updated, and the `allPegsCheckbox` updater must be registered as an observer of that notification to update itself in cases when the user clicked on one of the other checkboxes in the group. Cocoa does not provide a built-in means to suspend registration of notification observers on a per-selector basis, so you have to resort to a state variable to jury-rig your own routine to disable the effect of the notification on this one selector temporarily. It may be a good idea to mark this section to be

revisited later in case better ideas come to mind.

In the header file `MyWindowController.h`, declare the state variable after the existing outlet declarations:

```
NSControl *controlUpdatingDisabled;
```

This variable is typed to hold an `NSControl` object, so that any method testing its value can determine whether a specific user control's updating has been disabled. A `nil` value will indicate that no control's updating is currently disabled. The variable need not be initialized explicitly, because Objective-C will initialize it to `nil`, which should be the default value.

Also declare its accessor methods in the header file `MyWindowController.h`, at the end of the Accessor methods and conveniences section:

```
- (void)setControlUpdatingDisabled:(NSControl *)value;
- (NSControl *)controlUpdatingDisabled;
```

In the source file `MyWindowController.m`, define the accessor methods at the end of the `Accessor methods and conveniences` section:

```
- (void)setControlUpdatingDisabled:(NSControl *)value
{
    controlUpdatingDisabled = value;
}

- (NSControl *)controlUpdatingDisabled {
    return controlUpdatingDisabled;
}
```

Now, in the header file `MyWindowController.h`, declare `disableControlUpdating:` and `enableControlUpdating`, the methods that will be invoked in one or more action methods, as shown below, at the top of the `Interface management` section. You create and use these two additional methods in case a better way to disable and enable control updating is found later. If so, you will be able to redefine these methods and eliminate the state variable and its accessors.

```
// View update utilities

- (void)disableControlUpdating:(NSControl *)control;
- (void)enableControlUpdating;
```

Define these two methods in `MyWindowController.m`, as follows:

```
// View update utilities

- (void)disableControlUpdating:(NSControl *)control {
    [self setControlUpdatingDisabled:control];
}

- (void)enableControlUpdating {
    [self setControlUpdatingDisabled:nil];
}
```

In the `allPegsAction:` action method in `MyWindowController.m`, invoke these two methods to bracket the calls to the data accessor methods for the other three checkboxes in the group, so that this section of the action method reads as follows:

```
[self disableControlUpdating:sender];
[[self mySettings] setTrianglePegsValue:newState];
[[self mySettings] setSquarePegsValue:newState];
[[self mySettings] setRoundPegsValue:newState];
[self enableControlUpdating];
```

Finally, rewrite the `updateAllPegsCheckbox:` notification method in `MyWindowController.m` to test the state of the new state variable, like this:

```
- (void)updateAllPegsCheckbox:(NSNotification
*)notification {
    if ([self controlUpdatingDisabled] != [self
allPegsCheckbox]) {
        [self updateMixedCheckbox:[self
allPegsCheckbox] setting:[self
wantAllPegsCheckboxState]];
    }
}
```

5. The graphical user interface must be updated when a document is created or opened. As noted in instruction 2.j., above, you must therefore update the visible state of the `allPegsCheckbox` control. In `MyWindowController.m`, add the following call at the end of the `updateWindow` method:

```
[self updateAllPegsCheckbox:nil];
```

6. In addition, you must set up the `allPegsCheckbox` control to serve as a mixed-state checkbox. By

default, checkboxes are two-state; in response to an attempt to set the state of a two-state checkbox to the mixed state, it displays as checked, which is not what you want here. You will set up the `allPegsCheckbox` control as a mixed-state checkbox in the `windowDidLoad` method instead of the window controller's `init` method, because you can't be sure the checkbox is ready to be set up at initialization time. Add the following line just after the call to `super` in the `windowDidLoad` method of `MyWindowController.m`:

```
[[self allPegsCheckbox] setAllowsMixedState:YES];
```

7. There is one other thing you should do here, in anticipation of adding a number of additional user controls as the Vermont Recipes application grows in size and functionality. Many of the controls in this or any other application are used to set Boolean values, and those Boolean values need to be saved and retrieved from persistent storage. Initially, you wrote your storage routines so that they save Booleans as integers. As previewed in instruction 2.i, above, however, it may be more convenient to save them as human-readable "`YES`" and "`NO`" strings. This requires changing the `convertToDictionary:` and `restoreFromDictionary:` methods in MySettings so that they work with strings representing Boolean values.

   While this could be easily enough done by changing the code in those two methods, you will use a much cleverer and ultimately more efficient technique here, implementing an Objective-C category. You will create a new header file and a new source file declaring the category VRStringUtilities, which will add two new methods to Cocoa's built-in NSString class, one of them a class method that returns an NSString object equivalent to a Boolean value passed to it as a parameter, and the other an instance method that returns a C Boolean value equivalent to its string value. Once this category is available, your MySettings class can import it and call the two new methods as if they were built-in NSString methods.

   A note about categories

   Categories are a powerful feature of the Objective-C language. Categories allow you to extend and enhance the functionality of any other class, even if you don't have access to its source code, by adding or overriding methods already implemented in the class. When all you want to do is add a few methods to an existing class, categories may be a good substitute for subclassing the original class. New methods implemented in a category become available to all classes that import the category, and they are indistinguishable at run time from methods implemented in the original class. You can add both class methods and instance methods in a category.

   Categories are often declared and defined within header and source files for other custom classes, usually because the new methods in the category relate to the class in whose files it is declared. Multiple categories can even be declared on the original class simply as a device to break the class into convenient topical sections. Categories can also be used to break a single class into separate files to make it easier to manage; this is particularly convenient for managing a large, complex class.

You can also declare a category in a file of its own. For example, the category on NSString that you will create here is declared in separate header and source files, which could serve as the beginnings of a reusable custom string library for your private use.

a. First, you must create the two new files. You have done this before several times, by now, so the details needn't be belabored. In summary, choose File > New File… in Project Builder to create the source and header files for the new category, and name them **`NSString+VRStringUtilities.h`** and **`NSString+VRStringUtilities.m`**, respectively.

b. These are rather specialized files, so it might be useful to create a subgroup called "Categories" in the Groups & Files pane in the project window to hold these and any other categories you might create. Select the Classes group, then choose Project > New Group. A new folder icon appears nested under the Classes folder icon. Name the new subgroup **`Categories`**, then drag the two new files into it (the actual files will remain at the root level of your project folder in the Finder).

c. In `NSString+VRStringUtilities.h`, make sure the import directive imports `<Cocoa/Cocoa.h>`.

d. The interface declaration should be changed to read `@interface NSString (VRStringUtilities)`. This specifies that you have created the category VRStringUtilities adding functionality to the NSString class. You cannot declare new instance variables in a category, so there are no curly braces.

e. Declare the two new NSString methods, following the model of a number of existing NSString methods, as shown below. Notice that the first is a class method, like other NSString `stringWith...` methods, as denoted by the leading plus sign.

```
+ (id)stringWithBool:(BOOL)value;
- (BOOL)boolValue;
```

f. In `NSString+VRStringUtilities.m`, change the implementation declaration to read `@implementation NSString (VRStringUtilities)`.

g. Define the two new NSString methods as follows:

```
+ (id)stringWithBool:(BOOL)value {
    return [NSString stringWithString:(value) ? @"YES"
: @"NO"];
}

- (BOOL)boolValue {
    return ([self isEqualToString:@"YES"]) ? YES : NO;
}
```

h. It is that easy to add functionality to any existing class, even without having access to its source code. This new NSString functionality will be available to you everywhere in Vermont Recipes that you choose to import the category. Here, you want to use the new methods in MySettings, so add the following line to the source file `MySettings.m` after the existing import statements:

```
#import "NSString+VRStringUtilities.h"
```

i. Now all that remains is to use the new methods. In `MySettings.m`, return to the `convertToDictionary:` method and replace the statements you added in instruction 2.i., above, with the following:

```
[dictionary setObject:[NSString stringWithBool:[self
trianglePegsValue]] forKey:trianglePegsValueKey];
[dictionary setObject:[NSString stringWithBool:[self
squarePegsValue]] forKey:squarePegsValueKey];
[dictionary setObject:[NSString stringWithBool:[self
roundPegsValue]] forKey:roundPegsValueKey];
```

And replace the lines you added at the end of the `restoreFromDictionary:` method with the following:

```
[self setTrianglePegsValue:[[dictionary
objectForKey:trianglePegsValueKey] boolValue]];
[self setSquarePegsValue:[[dictionary
objectForKey:squarePegsValueKey] boolValue]];
[self setRoundPegsValue:[[dictionary
objectForKey:roundPegsValueKey] boolValue]];
```

j. As an exercise, change the statements that save and retrieve the `myCheckBoxValue` variable in the same way.

8. Before you can run the revised application, you must inform the nib file of the new outlets and actions you have created in the code files, then connect them to the new checkboxes.

a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two Vermont Recipes 2 header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

b. Select the Instances tab and Control-drag from the File's Owner icon to each of the four new checkboxes in turn and, each time, click its outlet name then click the Connect button in the Connections pane of the File's Owner Info palette.

c. Control drag from each of the four new checkboxes to the File's Owner icon in turn and, each time, click the target in the left pane and the appropriate action in the right pane of the Outlets area of the Connections pane of the NSButton Info palette, then click the Connect button.

9. Compile and run the application to test the interactions among the four new checkboxes. Explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly.

   Also, launch the PropertyListEditor application that comes with Cocoa, open a file saved by Vermont Recipes, and verify that the Boolean values appear as "`YES`" and "`NO`" strings.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 1: Create the project using Project Builder

Project Builder, commonly referred to as "PB," is the Mac OS X Integrated Development Environment (IDE) supplied by Apple for building Cocoa applications.

Documentation

The best place to begin reading about Project Builder is *PBOverview*, a very brief welcome note on your computer at */Developer/Documentation/ReleaseNotes/PBOverview.html*. (Later, you will also want to read *Project Builder Build Settings* at */Developer/Documentation/ReleaseNotes/PBBuildSettings.html*, but for now it is far too advanced to be of any use to you.)

Also read Project Builder Help in the Developer Help Center in the Apple Help Center, and read the Project Builder *Release Notes* that appear automatically in the main pane when you launch Project Builder. These and several other documents relating to the use of Project Builder are available directly from Project Builder's Help menu. Example projects with which to practice your skills are found on your computer at */Developer/Documentation/DeveloperTools/ProjectBuilder*. *Learning Cocoa* (O'Reilly, 2001) contains extensive instruction on the use of Project Builder in the context of these examples.

For a fuller understanding of Project Builder, you should read the old NextStep/OpenStep *Tools & Techniques Book*. It contains very detailed step-by-step instructions on the use of

Project Builder that are still useful. The book is on your computer at */Developer/Documentation/Cocoa/DevEnvGuide/Book/Tools&TechniquesBook.pdf*.

For general information about Project Builder and other developer tools, read *Developer Tools Overview* at */Developer/Documentation/DeveloperTools/DevToolsOverview.html*. A convenient roadmap appears at */Developer/Documentation/DeveloperTools/devtools.html*.

The first step in developing a Cocoa application typically is to set up the project files.

1. Launch Project Builder.
2. Choose File > New Project…. The New Project Assistant opens, listing a large number of templates with which you can start to develop an application, bundle, framework, kernel extension, or tool using the Cocoa, Java, or Carbon frameworks, or, for a tool, C++.
3. In the New Project Assistant, click the Cocoa Document-based Application template, under the Application heading, to select it. The Next button is enabled.
4. Click the Next button. The next panel of the New Project Assistant appears, in which you can name the project and specify its location. The current user's home directory is provided as the default location.
5. Type **`Vermont Recipes 1`** in the Project Name text box.
6. Click the Set… button to set the project's location, which may be easier for you than typing it. A sheet is presented in which you can navigate to any folder. Select the folder where you keep development projects (your home Documents folder is preselected for you), and click Choose. The sheet closes and the Finish button in the New Project Assistant is enabled.
7. Click the Finish button. The New Project Assistant closes and, after a pause, your project window opens in Project Builder under the name `Vermont Recipes 1.pbproj`.
8. Click the disclosure triangle next to the Classes folder icon in the Groups & Files pane of the project window, on the left, to expand it and see that the Classes topic holds two source files, `MyDocument.h` and `MyDocument.m`. Drag the border between the left and right panes to the right to see the full file names, if necessary. These files were created for you by Project Builder. They are templates containing code to get you started on your new Cocoa document-based application.
9. To see the text of the template header file, click `MyDocument.h` once to select it. The text of the header file appears in the main pane of the project window, if your Project Builder preferences have been left at their default values.

10. To see the text of the template source file, double-click `MyDocument.m`. The text of the source file appears in a new window. Opening separate windows by double-clicking the file name allows you to view multiple files side-by-side and to drag text among them. If you like, you can set a Project Builder preference so that source files always open in separate windows. Close the `MyDocument.m` window for now.

11. Click the disclosure triangle next to the Resources folder icon to see that it contains several items, including one called `MyDocument.nib`. `MyDocument.nib` is the nib file you will work with in Interface Builder in *Step 2* of this Recipe. Click the other disclosure triangles to see what they contain. The `main.m` file in Other Sources contains a standard C main function, which you will rarely need to modify for a Cocoa application. The Frameworks group contains a link to the Cocoa umbrella framework and to the Foundation and AppKit frameworks; you may add other frameworks here if your application requires them. The Products group is where your built application will reside, if your Project Builder preferences have been left at their default values.

12. Bring the Finder to the front and open the `Vermont Recipes 1` folder to examine its contents. This is the standard folder structure created for you by the Project Builder Cocoa Document-based Application template. In addition to the two MyDocument files and the `main.m` file, you see an `English.lproj` folder where nib files and other localizable information is kept. You also see the project file itself, `Vermont Recipes 1.pbproj`. This is actually a bundle; you can examine its constituent parts by choosing the contextual menu's Show Package Contents command. Notice that the folder structure in the Finder bears no relationship to the folder structure in the Groups & Files pane of the project window. The Groups & Files pane can be reorganized in any way you find convenient, without moving any of the files or folders in the Finder.

You could rename `MyDocument.h`, `MyDocument.m` and `MyDocument.nib` at this point, if you like. However, the remainder of this Recipe refers to them by these names, so you will be able to follow along more easily if you leave them as they are. If you want to rename them anyway, click on one of them in the Groups & Files pane of the project window and choose Project > Rename to select its text for editing. You must use a menu command to enable editing of group (folder) and file names in the left pane of the project window, because, as you learned in instruction 10, above, double-clicking opens the file in a new window.

You will defer further setup of the project for now and turn directly to interface design.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.6 Create outlets, actions, and connections

The application needs to know how the various objects and classes you have created relate to one another, so that one object can send messages to another at run time. In this Step, you will create an "action" to be invoked by your Checkbox user control; and you will create "outlets" in some of the objects to designate other objects to which they can talk.

Start where you left off in *Step 2.5*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

2.6.1 Create an action

An action, as you learned in *Step 2.1*, can be viewed as a message sent by a user control to a target object. In Cocoa, the action is implemented as a method in the target object, which Cocoa will automatically invoke when the user changes the control's state, for example, by clicking it. In this Step, you will create an action method in MyWindowController to be invoked when the user clicks the Checkbox user control.

You learned in *Step 2.3* that the role of MyWindowController is to mediate between the data controlled by its associated document and the user controls in its associated window. Each user control in the main document window will invoke an action method in MyWindowController when the user changes the state of the control, so that MyWindowController can in turn tell the document to update its data structures to reflect the user's action. The application knows where to find the proper action method because you will wire them together using Interface Builder. At this point, you have only one interactive user control in the main window, so only one action method need be implemented.

1. In the `MyDocument.nib` window, select the Classes tab. An outline list view of all available classes appears.

2. Scroll to the bottom of the list, and click MyWindowController once to select it. The line on which MyWindowController appears is highlighted.

3. Click the action button in the second column of symbols in the MyWindowController line (the action button is the small circle with a cross in it, denoting the crosshairs of a scope aimed at a target). The MyWindowController line expands to show two sublists, one for Outlets and one for Actions. A built-in action is selected, `showWindow:`. Notice that the names of actions end with a colon, indicating that they are Objective-C methods that take a single parameter. All action methods take `sender` as a parameter, giving the target a means to identify which control sent the action message.

4. Choose Classes > Add action. A new action appears in the Actions sublist, named `myAction:`. Its text is selected, ready to be edited.

5. Type to rename it **`myCheckboxAction:`**. Include the trailing colon indicating that this is a method. Hit the Enter key to accept the new name.

6. Bring the main document window to the front and choose the Buttons pane of the tab view, if necessary, to show the Checkbox user control. Then hold down the Control key and drag from the checkbox toward the `MyDocument.nib` window. While you drag, a line is drawn from the checkbox and follows the moving mouse pointer. The `MyDocument.nib` window automatically switches to the Instances pane as your drag extends over it. Let go of the mouse button over the File's Owner icon in the `MyDocument.nib` window to complete the drag (drag to an edge of the window, if ncessary, to scroll the File's Owner icon into view). The NSButton Info palette automatically appears and comes to the front, with the Connections pane showing.

7. Click the `target` outlet in the left column of the NSButton Info palette once to select it, if necessary. Then click the `myCheckboxAction:` method once in the right column to select it. The Connect button is enabled. Finally, click the Connect button at the bottom of the Info palette. Your new connection appears in the bottom pane of the Info palette.

You will add code to the project in *Step 3.5* and *Step 3.6* to complete your work to implement this action.

2.6.2 Create outlets and connections

Some of your application's objects need outlets to other objects, in order to be able to send messages to them and obtain information from them. To finish wiring up the user action you created in *Step 2.6.1*, for example, you need an outlet in your main window's controller to its document, so that the window controller can tell the document to alter the data it controls and, later, to retrieve the value of the data. And you need an outlet from the window controller to the Checkbox user control in the main window, so the

window controller can tell the control to alter its appearance to reflect the state of the document's data.

It turns out, however, that Cocoa has already done some of this work for you.

2.6.2.1 An outlet from the window controller to the document

In *Step 2.6.1*, you created an action method, `myCheckboxAction:`, so that the Checkbox user control can tell the file's owner, MyWindowController, when the user has changed the state of the control. When a MyWindowController object receives such a message at run time, it must be able in turn to tell its associated document to change its data structures to reflect the user's action. The window controller must also at times be able to obtain data from the document, in order to alter the user interface to correspond to the state of the document and for other purposes. To do all this, MyWindowController must have a way to talk to MyDocument.

In fact, however, the window controller class already has a means to converse with its associated document. When you examine the header files and documentation for the NSWindowController class, you will find that it has a method, called `document`, which returns the window controller's associated document object. Recall that a document object is created at run time whenever the user requests a new document or opens an existing document. At the time of the document's creation, NSWindowController automatically sets up this method for you.

You will find that many Cocoa classes automatically provide such connections when you need them, so that you need not create an explicit outlet yourself. The only way you can know whether you need to create your own outlet is to become familiar with the Cocoa classes. A common beginner's error is to create custom outlets without realizing that they are already built into the Cocoa framework.

2.6.2.2 An outlet from the window controller to a user control

To complete the network of actions and outlets involving the Checkbox user control, you must create an outlet from the MyWindowController subclass to the control, so that the control can be told to change its appearance when appropriate. This outlet is not provided in the NSWindowController superclass because the Cocoa frameworks cannot know in advance that you would create this particular user control for your application.

1. In the `MyDocument.nib` window, select the Classes tab. An outline list view of all available classes appears.

2. Scroll to the bottom of the list, and click MyWindowController to select it.

3. Click the outlet button in the second column of symbols on the MyWindowController line (the outlet button is the small circle with two dots in it, denoting an electrical outlet). The MyWindowController line expands to show two sublists, one for Outlets and one for Actions.

4. Choose Classes > Add outlet. A new outlet appears in the Outlets sublist, named "myOutlet". Its text is selected, ready to be edited.

5. Type to rename it **`myCheckbox`** and hit the Enter key. Notice that no colon is needed, because this is not a method but an instance variable.

6. In the `MyDocument.nib` window, choose the Instances tab. Then hold down the Control key and drag from the File's Owner icon to the Checkbox user control in the Buttons pane of the main window's tab view. While you drag, a line is drawn from the File's Owner icon and follows the moving mouse pointer. Let go of the mouse button over the Checkbox user control to complete the drag. The File's Owner Info palette automatically appears and comes to the front, with the Connections pane showing and the `myCheckbox` outlet selected. The Connect button is enabled.

7. Click the Connect button at the bottom of the Info palette. The new connection appears in the bottom area of the Info palette.

You will add code to the project in *Step 3.4* to complete your work on this outlet.

2.6.3 Connect other outlets

You will need to connect a number of additional outlets to permit various objects to talk to one another. You decide what outlets and connections are needed by thinking out which objects must be able to control or obtain information from other objects. If you forget something, you can create more actions, outlets, and connections later. Here you will confirm or make several connections on built-in outlets.

1. MyWindowController already has a window outlet provided by Cocoa, so you don't need to create it. It is already connected to the Parent Window, too. To verify this, click in the Instances tab of the `MyDocument.nib` window, select the File's Owner icon, click the window connection in the bottom section of the Connections pane of the File's Owner Info palette, and see where the line leads. You made this connection in *Step 2.1*.

2. Control-drag to draw a connection from the Parent Window icon to the File's Owner icon, which now represents the window controller. When you complete the drag, the Window Info palette comes to the front. This time, select the existing `delegate` outlet and click the Connect button. This appoints MyWindowController the Parent Window's delegate, which will permit built-in Cocoa window routines to delegate various tasks to MyWindowController objects at run time. You will see an example of delegation in use later, in *Step 5.1*. Many AppKit classes have built-in delegate outlets; you will connect them to other objects at your option, depending on whether you want to take advantage of delegated functionality.

3. You should also connect the `initialFirstResponder` outlet in the Parent Window, so that the application will know, when a new window is opened, which one of its user controls has the "focus" or is "key." Control-drag from the Parent Window icon in the `MyDocument.nib` window

to the Checkbox user control in the Buttons tab view item of the main window, select `initialFirstResponder` in the Window Info palette, and click the Connect button.

4. Save your work in the nib file by choosing File > Save.

In the next Step, you will create the source files required to start programming the application.

Introduction > Contents > Recipe 1 > Step 2.6 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.7 Create the source files

You are now ready to generate source files. You could instead continue working in Interface Builder to add more user controls to the main window, and you could also use Interface Builder to create a user interface and connections for DrawContentView and additional drawers and windows. However, for didactic purposes, you will add additional controls later in source code using Project Builder, instead.

In this Step, you will use Interface Builder to generate source code for the MyWindowController subclass, because that code does not already exist in any Project Builder template. You will also generate source code for MyDocument, even though the Cocoa Document-based Application template has already provided starter code for that subclass, just to see what Interface Builder produces.

Eventually, when the code generation features of Interface Builder and Project Builder are fully integrated, you will be able to use Interface Builder to merge new outlets and actions created in Interface Builder into your existing Project Builder header and source files. Doing so will place stub declarations for the new outlets and actions into the existing project source files, without overwriting existing code. However, if you were to use Interface Builder's Classes > Create Files… command today, in Mac OS X 10.0, to create MyDocument source files, you would risk overwriting the existing MyDocument files, losing all their existing code. You will therefore generate code for those files in a different folder, in order to examine it. Portions of code generated in this fashion could be cut to the Pasteboard and pasted into an existing source file, if desired.

Start where you left off in *Step 2.6*. Launch Interface Builder and open `MyDocument.nib`, if necessary.

1. In the `MyDocument.nib` window, select the Classes tab. An outline list view of all available classes appears.

2. Scroll to the bottom of the list and click MyWindowController to select it.

3. Choose Classes > Create Files…. A sheet opens, letting you select a location in which to create the files. You also see two checkboxes in which to confirm that you want to create `MyWindowController.h` and `MyWindowController.m`, and they are checked by default. If the project is still open in Project Builder at this point, you also see a checkbox in the Insert into targets area in which to confirm that the files should be inserted into the `Vermont Recipes 1` target; normally, you would leave this checkbox checked, but for present purposes uncheck it if you see it. Leave the checkboxes for creating the two source files checked, click the `Vermont Recipes 1` folder to confirm that this is where the files should be placed, and click the Choose button. As you will see in instruction 6., below, the two new files are created in the `Vermont Recipes 1` folder.

4. Scroll to the upper part of the Classes list in the `MyDocument.nib` window and click MyDocument.

5. Choose Classes > Create Files… again. A sheet opens, letting you select a location in which to create the files, and checkboxes to confirm that you want to create `MyDocument.h` and `MyDocument.m`. In this case, however, you see that the `Vermont Recipes 1` folder already contains files named `MyDocument.h` and `MyDocument.m`, where you created them earlier, and you don't want to overwrite them.

   Out of curiosity, however, it would be interesting to see what Interface Builder generates. Uncheck the Insert into Project Builder checkbox, use the navigation menu to select your home `Documents` folder, and click the Choose button. (If an alert appears telling you that these files already exist, you must have selected the `Vermont Recipes 1` folder by mistake; cancel immediately to avoid overwriting your files.) If all went well, use the Finder to navigate to your `Documents` folder, where you find the new `MyDocument.h` and `MyDocument.m` files. Open them by double-clicking them or by dragging them onto the Project Builder icon in the Dock. When they open in Project Builder, you will see that these new files contain no code. This is because you created no outlets or actions for the MyDocument class in Interface Builder. Using the Finder, drag both of them to the trash. Be sure to leave the original `MyDocument.h` and `MyDocument.m` files in the `Vermont Recipes 1` folder.

6. Using the Finder, navigate to the `Vermont Recipes 1` folder. You will find the two new MyWindowController files you just created. Open them. You see that they are very simple, with stubs for the action method and each of the outlets you created in *Step 2.6*, but not much else. Don't be discouraged; the nib file contains essential additional information behind the scenes that will greatly simplify the work remaining.

In the next Step, you will finish merging the source files into the project.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe01_step02_07.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 2: Design and build the graphical user interface using Interface Builder

2.8 Merge the source files into the project

Assuming you did not check any project entry in the Insert into targets pane in instruction 3. of *Step 2.7*, you now have to merge the two new MyWindowController files into the project.

Start where you left off in *Step 2.7*. Launch Project Builder, if necessary.

1. To merge MyWindowController into the project:

    a. In the Finder, confirm that `MyWindowController.h` and `MyWindowController.m` were saved in the main `Vermont Recipes 1` folder when you generated them in Interface Builder. If you saved them somewhere else in the previous Step, drag them into the folder now. It is important to place files in their proper locations in the Finder before adding them to the project.

    b. In Project Builder, choose Project > Add Files…. A sheet opens in which you can select files to add.

    c. In the sheet, Navigate to the `Vermont Recipes 1` folder. Select `MyWindowController.h` and `MyWindowController.m`, holding down the Shift key to select both at once. Click the Open button. Another sheet opens, in which you can set options.

    d. In the options sheet, you normally check the Copy into group's folder (if needed) checkbox

to ensure that the files are moved into the project folder in the Finder. You have already done this manually in Step 1.a., above, but it does no harm to get in the habit of checking this checkbox. The setting of the other controls in the options sheet doesn't matter at this stage; leave them as you find them. Click the Add button. Both files are added to the left pane of the project window, entitled "Groups & Files."

e. In the project window, drag both files into the Classes folder icon in the Groups & Files pane of the project window, if necessary. This does not move the files on disk or in the Finder, but it does organize the Groups & Files pane according to the conventions of Cocoa development. You are free to rearrange the Groups & Files pane in any way that you find convenient.

You are just about ready to begin coding the application.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

3.1 Set up the project target and resources

At some point, you must set up the application's Finder-related information and other settings, such as its `CFBundlePackageType`, `CFBundleSignature`, and `CFBundleDocumentTypes`. It is a good idea to get these out of the way up front.

Documentation

The *Software Configuration* chapter of *Inside Mac OS X: System Overview* in */Developer/Documentation/SystemOverview/SystemOverview/index.html* explains the function and format of the `Info.plist` and `InfoPlist.strings` files found in every application package and other bundles. It includes an explanation of each item in these files. Also read the *Bundles* and *Application Packaging* chapters for information about localizable strings and other resources. Be sure to check the Cocoa documentation web site for the latest version of *System Overview*.

The *Info Property List Release Note* in */Developer/Documentation/ReleaseNotes/InfoPlist.html* contains important information about new features of the Info.plist file in Mac OS X 10.0. Also search on "Info.plist" or any of the individual items in it in the Developer Help Center in the Apple Help Center for additional information.

For more information on localizable strings, read the *Internationalization* chapter of

System Overview.

Launch Project Builder and open the project, if necessary.

3.1.1 Set up the Application Settings

The settings you will provide in the Application Settings pane of the Targets tab of the project window will become the `Info.plist` file in the application's bundle when you use Project Builder to build your application. In Mac OS X 10.0, Project Builder creates the application as a so-called "new-style" bundle, automatically saving the `Info.plist` file in XML property list format. In earlier versions, the application was saved as a so-called "versioned" bundle; the `Info.plist` file was saved as a plain text property list, and some of the information was also saved in a `versions.plist` file in the application bundle. The `Info.plist` file is used by Cocoa for a variety of purposes, including providing the short application name that appears in the menu bar when the application is running; telling the system where the application and document icons are located; providing the version, copyright and other strings used in the Get Info dialog, the about box, and other dialogs and alerts; and providing recognized document types used to tell the desktop to open the application when the user double-clicks one of the application's document icons. Project Builder now also places the type and creator code for your application into a file named `PkgInfo`, which is used by the Finder to cache this information for performance reasons.

1. In the project window, click the Targets tab. The Targets pane slides into view in the left pane.

2. Click Vermont Recipes 1 in the Targets pane, on the left. The main pane of the project window shows several tabs.

3. Select the Application Settings tab and click the Expert button. You will change only some of the default settings found here. (The Simple button is left as an exercise for the reader; note that there are Help tags associated with some of the fields to help you identify their purpose, and the *Setting Bundle Options* subtopic in the *Target Options* topic in Project Builder Help contains additional information).

4. Click the disclosure triangles to expand `CFBundleDocumentTypes`, then expand element 0, then expand `CFBundleTypeExtensions` and `CFBundleTypeOSTypes`.

    a. For `CFBundleTypeExtensions 0`, double-click "????" to select the field for editing, then replace its contents with your document's four-character type. For Vermont Recipes 1, type **`VRd1`** ("VR" stands for Vermont Recipes and "d" stands for document, in the locution we use here).

    b. For `CFBundleTypeName`, replace "DocumentType" with **`Vermont Recipes 1 Document Format`**. Later, in *Step 4.2.1*, you will code a variable using this same string.

    c. For `CFBundleTypeOSTypes 0`, replace "????" with **`VRd1`**.

d. Leave the rest of the `CFBundleDocumentTypes` as you find them. They are correct for the application you are building.

5. Change `CFBundleSignature` from "????" to your application's creator. For Vermont Recipes 1, type **`VRa1`** ("a" stands for application). Note that this has not been registered with Apple Computer. For a real application, you should register your creator with Apple.

6. Change `CFBundleVersion` to **`1.0.0d1`**, since this is the first development build of what will become version 1.0.0 of the application. Cocoa knows how to deal with version strings using this traditional Macintosh versioning format.

7. Click the New Sibling button. Type over the New Item name to create an item named **`CFBundleIdentifier`**. Click on empty space in the Application Settings pane to let the new item move to its alphabetical place. Double-click in the Value column opposite the new item to select it for editing, then type **`com.stepwise.VermontRecipes.VermontRecipes1`**. This is the domain for the application, which Mac OS X uses for various purposes that you will learn about later, including locating the application's preference files.

8. All the other settings are correct for now, so click the Files tab of the project window, and choose File > Save.

3.1.2 Set up InfoPlist.strings

The `InfoPlist.strings` file provides English-language localizations for some of the settings in the `Info.plist` file that you just finished setting up. Localization files are saved as plain text property lists, with comments to help localization contractors identify the use and purpose of the strings. Some of these strings are used, for example, in the application's about box.

1. Click the Files tab in the project window to return to the Groups & Files pane. Expand the Resources group in the Groups & Files pane and click `InfoPlist.strings`. Several settings appear in the main pane of the project window, most of which need to be edited.

    a. Change `CFBundleName` to **`Vermont Recipes 1`** if it is not already set to that string.

    b. Change `CFBundleShortVersionString` to **`Vermont Recipes 1 1.0.0d1`**.

    c. Change `CFBundleGetInfoString` to **`Vermont Recipes 1 1.0.0d1, Copyright \U00A9 2000-01 Bill Cheeseman.`**.

    "\U00A9" generates the standard © symbol.

    d. Change `NSHumanReadableCopyright` to **`Copyright \U00A9 2000, Bill Cheeseman. All rights reserved.`**.

e. There is a new `InfoPlist.strings` requirement introduced in Mac OS X 10.0, described in the *Info Property List Release Note* referenced above. The `CFBundleTypeName` entry you created for the `Info.plist` file in *Step 3.1.1*, above, is now displayed by the Finder as the "Kind" string for documents. To ensure that an appropriate string is displayed by the Finder for Vermont Recipes 1 documents, add this entry at the end of `InfoPlist.strings`:

```
"Vermont Recipes 1 Document Format" = "Vermont Recipes 1 document";
```

Note the quotation marks around the key on the left as well as the value on the right.

2. Choose File > Save.

3.1.3 Set up Credits.rtf

The contents of `Credits.rtf` are shown in the application's about window. This didn't work in Mac OS X Public Beta, but it was fixed for Mac OS X 10.0.

1. Click `Credits.rtf` in the Groups & Files pane of the project window and make any changes you like. To view the settings we use, click `Credits.rtf` in the `Resources` folder of the downloadable project files for Vermont Recipes 1.

2. Choose File > Save.

3.1.4 Set up Localizable.strings

A `Localizable.strings` file should be added to the project for each localized or language-specific version of the application. It contains key-value pairs specifying string values for any localizable string used in the application, for example, for menu item names, button names and the like. "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, by convention, the ".strings" extension. All `.strings` files for a given language belong in that language's `.lproj` folder. The files take the form of plain text property lists with comments to assist localization contractors. Here, you will create a `Localizable.strings` file for the `English.lproj` folder.

Specifying strings in localizable, or "internationalized," 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 the `@"this is a string"` form to provide fixed user-viewable text, you should instead use the `NSLocalizedString()` function, or, if you have `.strings` files with names other than "Localizable.strings," the `NSLocalizedStringFromTable()` function. These are convenience functions defined in the NSBundle header in the Foundation framework, which call NSBundle's `localizedStringForKey:value:table:` method on the application's main bundle. In the `NSLocalizedString()` function, you pass two parameters specifying the key and a comment string

explaining what you are doing. The comment is not used but exists only to force you to document your code; it should be descriptive and should be repeated verbatim as a comment in the `Localizable.strings` file or similar file to assist your localization contractor. In the `NSLocalizedStringFromTable()` function, 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. For examples of the use of the `NSLocalizedString()` function to name menu items and to compare names of menu items, see *Step 5.2* and *Step 6*.

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

1. Choose File > New File… in Project Builder.
2. Select "Empty File" at the top of the list and click the Next button.
3. Set the name of the file to **`Localizable.strings`**, set its location to the `English.lproj` folder in the project folder, and click the Finish button.
4. In the Groups & Files pane, drag the new `Localizable.strings` item into the Resources group, if necessary.
5. If you wish (this is not required), click `Localizable.strings` to bring the new, empty file into the right pane, and type a heading such as **`/* English strings for Vermont Recipes 1 */`**. From now on, whenever you insert the `NSLocalizedString()` function or the `NSLocalizedStringFromTable()` function into your code, you must also return to the `Localizable.strings` file and provide a suitable key-value pair and comment to match.
6. Choose File > Save.

In the next series of Steps, you will begin coding the application.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

You are finally ready to begin writing code. The application you will create in these Recipes is, as you already know, a document-based application. If you have not already done so, you should now read some background documentation on basic concepts.

> **Documentation**
>
> A good technical overview of what you have to do to create a document-based application appears in the *Document-Based Application Architecture* section of the *NSDocument* class reference document, available in the Cocoa section of the Developer Help Center in the Apple Help Center.
>
> You should also read *Application Design for Scripting, Documents, and Undo*, located on your computer at */Developer/Documentation/Cocoa/ProgrammingTopics/AppDesign/AppDesign.html*. It is also available in PDF form in the same folder.

3.2 Import the Cocoa umbrella framework

Older versions of Interface Builder and Project Builder generated templates that imported the Application Kit framework. However, *Inside Mac OS X: System Overview* recommends that Cocoa applications import the Cocoa umbrella framework, instead. The Cocoa framework imports both AppKit and the Foundation framework, and it may from version to version import other headers that may be required for Cocoa

development. For example, in Public Beta it imported `AppKitScripting.h` for AppleScript support.

The current versions of the developer tools import the Cocoa umbrella framework, so this Step is now only of historical interest.

Launch Project Builder and open the project, if necessary.

1. Open each of the header files in the Classes group in the Groups & Files pane of the project window by clicking them in turn.
2. In each, if necessary, replace the line `#import <AppKit/AppKit.h>` with the following:

```
#import <Cocoa/Cocoa.h>
```

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

3.3 Replace the window routines provided by the Project Builder template

The Cocoa Document-based Application template contemplates an application in which only one window is opened for each document. Accordingly, the first method in `MyDocument.m` overrides NSDocument's `windowNibName` method, which is designed to work with such an application.

However, the template includes a comment specifically advising to delete `windowNibName` and instead override `makeWindowControllers` in an application like Vermont Recipes, which subclasses NSWindowController and where each document can have more than one kind of window. In this Step, you will do this.

The Cocoa documentation also makes clear that you need not implement `windowControllerDidLoadNib:`, although you can do so if there is something you want to do just after the nib file is loaded. You don't need `windowControllerDidLoadNib:` at this point, so you will delete it for now.

Launch Project Builder and open the project, if necessary.

1. Click the source file `MyDocument.m` in the Groups & Files pane of the project window. The full text of `MyDocument.m` appears in the main pane.

2. Add this line to `MyDocument.m`, after `#import "MyDocument.h"`:

```
#import "MyWindowController.h"
```

Without this, the override of `makeWindowcontrollers` in instruction 4., below, would generate compiler warnings because the compiler wouldn't see a declaration for that method.

3. Delete the first two methods from `MyDocument.m`, `windowNibName` and `windowControllerDidLoadNib:`, which were generated by the Cocoa Document-based Application template but aren't appropriate for the Vermont Recipes application.

4. Replace the deleted methods with the following:

```
// Window management

- (void)makeWindowControllers {
    MyWindowController *controller =
[[MyWindowController allocWithZone:[self zone]] init];
    [self addWindowController:controller];
    [controller release];
}
```

You do not need to declare this method in the header file `MyDocument.h`, because it will not be called explicitly from your code. It merely overrides a built-in NSDocument method that will be invoked for you at the proper time by the Cocoa frameworks. Whether to declare override methods in the header file is a matter of personal preference; many programmers add matching declarations for all methods that are defined in the source, or implementation, file, because the header file can then easily be used as a table of contents for the source file. Here, we choose not to declare override methods, on a general principle of parsimony. This also makes it easy to see with a glance at the header file all of the custom methods you have written, because you aren't forced to try to remember whether a particular method signature exists in the built-in Cocoa frameworks. More importantly, if you distribute your header files without the source files, it allows you to change what methods are overridden in the implementation without requiring your customers to work with new headers.

This is your first encounter with the standard technique to instantiate and initialize objects using Objective-C in Cocoa. The first line in `makeWindowControllers` combines the allocation of memory for the new object with an invocation of its initialization method, all in a single line of code. Depending on the object, it may have a variety of initialization methods, and it is important to invoke and obtain a return value from the one that is relevant to the purpose at hand. Usually, that will be the so-called "designated initializer." Invoked here is the designated initializer for MyWindowController, its `init` method. You will see in a moment that its `init` method will call a built-in method that initializes the window controller and makes it the owner of the nib file.

As instructed in *Step 2.3*, you did not instantiate a window controller in Interface Builder. Now you have closed this gap by placing code in your MyDocument class that will be invoked automatically by Cocoa to instantiate and initialize a window controller every time Cocoa creates a new document object in response to user commands. The second line inserts the new window controller object into the built-in NSDocument `windowControllers` array.

Notice that this method allocates memory for a new object, a window controller. This is not the place to delve too deeply into the difficult subject of reference counting and the autorelease pool, and when to use `retain`, `release` and `autorelease` in your code. Take it on faith for now that the memory for the window controller object that is allocated here was automatically retained by the invocation of the `allocWithZone:` method. It is therefore said to be "owned" by you; that is, you are responsible for releasing it when the application has no further use for it.

Here, this is accomplished right away, in the last line of the same method that allocated the object. It is quite common to release an object in the same method in which you allocate it. It is safe to release it here, even though it is being added to the document's `windowController` array and the window will remain open. This is because, as a general rule, a method like `addWindowController:` that adds an object to a "collection" (here, an array) does its own retain on the object. You do not "own" the object once it is inserted into the array and, in a document-based application like Vermont Recipes, it will therefore be released again automatically by Cocoa when the window is closed. Some of this is described in the *Window Closing Behavior* section of the *NSWindowController* class reference document.

Confused? So is almost everybody at this stage, so don't worry about it. Basically, invoking an object's retain method simply increments its reference counter by one. Invoking the object's release method does not actually deallocate the object; it only decrements its reference counter by one. As long as its reference counter is greater than zero, the object survives. Very shortly after its reference counter becomes zero, probably when the current iteration of the application's main run loop terminates, the memory will be deallocated. In the method we are discussing here, when the window closes, the two retains will have been balanced by the two releases, and the object will go away, thus avoiding a memory leak.

Documentation

To learn more about memory allocation and deallocation, reference counting, `retain`, `release`, and `autorelease`, read Very Simple Rules for Memory Management in Cocoa, a 2001 article by Mmalcolm Crawford, Memory Management with Cocoa/WebObjects, a1999 article by Manulyengar, and Hold Me, Use Me, Free Me, a 1997 article by Don Yacktman, all on the Stepwise site.

5. Click the source file `MyWindowController.m` in the Groups & Files pane of the project

window. The text of `MyWindowController.m` appears in the main pane.

6. Whenever a new window controller is created, it must initialize itself. If it allocates memory for an object, the window controller must also release that memory when the window controller is destroyed. In deciding whether a subclass needs a method to release memory, you must take into account whether it allocates memory, as well as other issues.

   In `MyWindowController.m`, insert the following two methods after `@implementation MyWindowController`:

```
// Initialization

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

- (void)dealloc {
    [[NSNotificationCenter defaultCenter]
removeObserver:self];
    [super dealloc];
}
```

   In instruction 4, above, you arranged for a new window controller to be allocated and initialized every time a new document is created. The `init` method that you invoked in that instruction is defined here. It in turn invokes a method implemented by its superclass, NSWindowController, called `initWithWindowNibName:`. (Note that the name passed to `initWithWindowNibName:` is the name of the MyDocument.nib file, without the trailing ".nib".) The `initWithWindowNibName:` method has the effect, among other things, of making the new window controller the owner of the nib file. By loading the nib file in this way, the application acquires knowledge of all the classes and subclasses, the windows and the user controls that you created in Interface Builder in *Step 2*.

   The syntax used to pass the name to the `initWithWindowNibName:` method, `@"MyDocument"`, is the standard Objective-C technique to pass fixed NSString objects to an application. You will make frequent use of this technique. (You will shortly see how to resolve localization issues raised by the use of fixed, or hard-coded strings in your code. This is not an issue here, however, because the string is the name of a source file that will have the same, fixed name in every country.)

   The override of NSWindowController's `dealloc` method takes care of a notification center issue that you don't yet need to understand (see instruction 6 of *Step 5.3*), then it calls its superclass's `dealloc` method to release any memory it allocated.

Now, every time your application opens a new document, it will automatically create a corresponding window controller instantiated from the MyWindowController subclass, which will in turn open that document's main window. When the document is closed, the window controller will be deallocated.

7. You know from the application specification that every document will optionally be able to open one or more separate windows. These separate windows are ancillary to the main document window and they will contain information that is not identical to that displayed in the main window. It will therefore be appropriate for them to be closed automatically when a document's main window is closed. That is, the document itself should close when its unique main window is closed, even if ancillary windows from the same document are also open. You should take care of this detail now. In your new `init` method in `MyWindowController.m`, you will revise the `init` method and add a line invoking an NSWindowController method to reverse the NSWindowController default setting on this point. Doing this requires a slight change to the structure of the `init` method that you just wrote in instruction 6., above. Replace the `init` method with the following new version:

```
- (id)init {
    if (self = [super
initWithWindowNibName:@"MyDocument"]) {
        [self setShouldCloseDocument:YES];
    }
    return self;
}
```

An initialization method must always return an object, usually the super's `self`, if it succeeds; if it fails, it should always return `nil`. The test at the beginning of this method ensures that initialization of the superclass has succeeded before an attempt is made to use the `setShouldCloseDocument:` method of `NSWindowController`. A `nil` value will have been returned if initialization of the superclass failed; the test is necessary because attempting to call a method of a `nil` object would, of course, cause an error. This statement is tricky; it combines in one line the assignment of the return value of the super's `initWithWindowNibName:` method to `self` with a test to see whether the result is `nil,` indicating a failure to allocate the super. Don't mistake this for a direct comparison of `self` with the result of the call to `[super initWithWindowNibName:@"MyDocument"]`, which would require the `==` operator and would always return false. Here, the assignment in parentheses is executed first; then, if `nil` was assigned to self, the `if` test evaluates the `nil` value as `NO` and `[self setShouldCloseDocument:YES]` is not invoked. Your `init` method then returns the `nil` value of `self`, as it should because the initialization of the superclass has failed.

8. It seems to be customary in some circles to declare an `init` method in the header file even though

it is an override method, but not to declare a `dealloc` override method. Others consider it silly to declare methods like `init` and `dealloc` that everyone knows are commonly overridden, while still others think it important to declare *every* method that is implemented. In *Vermont Recipes*, we will not declare `init` or `dealloc` methods in header files.

If you wish to follow the practice of declaring `init` methods, click the header file `MyWindowController.h` in the Groups & Files pane of the project window. The text of `MyWindowController.h` appears in the main pane. Add the following line to `MyWindowController.h`, after the `@interface MyWindowController` block (that is, outside the curly braces):

```
- (id)init;
```

If you wish to declare the `dealloc` method, too, you are left to your own devices.

You will not be reminded hereafter to save your work at the end of each step.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

3.4 Implement the user control outlet created in Interface Builder

In *Step 2.6.2.2*., you used Interface Builder to create an outlet from the `MyDocument.nib` file's owner, MyWindowController, to the Checkbox user control in the main document window. You must now see to it that this outlet is correctly implemented in Project Builder.

Launch Project Builder and open the project, if necessary.

1. Click the header file `MyWindowController.h` in the Groups & Files pane of the project window. `MyWindowController.h` appears in the main pane.

2. You see that the interface declares an object, `myCheckbox`, of generic object type `id`, with the `IBOutlet` prefix. This declaration was created by Interface Builder when you used its Classes > Create Files… command. If you were later to read `MyWindowController.h` back into Interface Builder to update the nib file, Interface Builder would recognize this as an outlet even without the `IBOutlet` prefix, because it is of type `id`. However, you will gain the benefit of stricter type checking at compile time if you change it to an object of type NSButton. When you do this, you must, for the benefit of Interface Builder, explicitly identify it as an Interface Builder outlet; you must therefore use the `IBOutlet` prefix.

   Delete the line `IBOutlet id myCheckbox;` in `MyWindowController.h` and replace it with the following lines:

```
@private
IBOutlet NSButton *myCheckbox;
```

You use the `@private` directive to ensure, theoretically, that any subclasses of `MyWindowController.h` will be denied direct access to the instance variable. It signals that they should instead use the accessor method that you will provide shortly. Declaring instance variables private in this fashion is optional, but if the directive is honored by subclassers, it gives you the ability in the future to change the way the checkbox object is coded without invalidating any subclasses that may come to exist. In reality, these variables can still be accessed by subclasses using Objective-C techniques, so the effect is more in the nature of moral suasion than full security.

You might think, from a first reading of the Objective-C documentation, that you need to add the line `@class NSButton;` before `@interface` in `MyWindowController.h` to tell the compiler that you intend to use that type in the source file. However, in this case you do not need such a line because your header file already imports the `Cocoa.h` umbrella header, which indirectly imports NSButton. You must either import a class into the header or, if that might create a circular reference, use `@class`, in order to avoid an undefined type error when compiling and linking your project, with respect to any type referenced in the header. If you use `@class` in a header file for a custom class, you must import the class in the source file if you use it there.

3. You created the outlet to the checkbox user control in the first place because you anticipated that your document object might have to send a message to the control to modify its visible state from time to time, or at least to read its state. For example, the application might have to check it or uncheck it in order to keep it synchronized with the underlying data, perhaps because a related menu command was used to set the underlying data. One way the application could do this would be to manipulate the user control's appearance directly, via its `myCheckbox` variable. However, to isolate your application's data representation from the code as much as possible, you should add an accessor method to be used to access the control, instead. The use of accessor methods for this purpose is a commonplace of Cocoa development.

    a. Add the following to `myWindowController.h`, immediately before the `myCheckboxAction:` method that was put there by Interface Builder:

    ```
    // Accessor methods and conveniences

    - (NSButton *)myCheckbox;
    ```

    You have noticed that we are adding descriptive headings to the source files as we go. These files will become very large before you are done with *Vermont Recipes*, so it is important to use an organizing principle like this to help you find your way around them. You needn't use our wording or organization; every programmer has a favorite technique. This would be a good time to place a heading like the following above the `myCheckboxAction:` method,

both in the header and the source file:

```
// Action methods
```

b. Switch to the source file `MyWindowController.m` and add the following lines before the `myCheckboxAction:` stub:

```
// Accessor methods and conveniences

- (NSButton *)myCheckbox {
    return myCheckbox;
}
```

Now, whenever you need to send a message to the window controller's associated checkbox object, you will be able to write something like `[[[self myWindowController] myCheckbox] doSomething]` from any object that has an outlet connected to the window controller.

Now is a good time to add a descriptive heading above the stub definition of the `myCheckboxAction:` method in the source file, just as you did in the header file in instruction 3.a., above.

Accessor methods are a standard feature of any Objective-C Cocoa application. Your code will be full of accessor methods that look like this in no time at all.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 3: Set up the project source files using Project Builder

3.6 Implement the action created in Interface Builder

Now that the MySettings object knows how to store and fetch the data value represented by the checkbox control, and now that the window controller can talk to the MySettings object, you can return to MyWindowController's action method and enable it to mediate between the data and the user interface. This is the role prescribed for a window controller in the MVC paradigm.

1. In the source file `MyWindowController.m`, for a first attempt, insert the following in the stub `myCheckboxAction` method provided by Interface Builder:

```
[[self mySettings] toggleMyCheckboxValue];
```

   Normally, an action method would use the reference to `sender`, a parameter that is always passed with an action method so that the action method can know what object initiated the action. But you didn't do that here. Here, you might think you can get away without referring to the state of the sender, on the premise that any time the user clicks a checkbox it always reverses state. On this assumption, the data value in memory could simply be toggled, too.

   However, for a variety of reasons, you should not do it this way. Among other things, in order to ensure that the data in MySettings does not fall out of sync with the state of the control in the user interface due to an error somewhere else, it would be safer and make debugging easier if you replace the statement you just typed with the following:

```
[[self mySettings] setMyCheckboxValue:([sender state]
== NSOnState)];
```

This new implementation of the action method reads the visual `state` of the user control object `sender`—which has already changed due to the user's having clicked it—then asks whether it is `NSOnState` and sets the data value accordingly.

Having done this, you should now delete the `toggleMyCheckboxValue` method from the MySettings header and source files. You no longer need it, since the `setMyCheckboxValue:` method is a safer and more general means to accomplish the same thing.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 4: Provide for data storage and retrieval

Your application must be able to save its data to storage, typically on a disk, and read it back from storage when a document is opened or reverted to a previously saved state. In this Step, you will add generic routines for data storage and retrieval to the application.

4.1 Initialize the data

Before coding the application's storage routines, you should set up the default value of the data represented by the checkbox user control when a new document is created from scratch. For now, you will simply initialize a new document so that the initial setting of the `myCheckboxValue` variable in MySettings is `YES`. In a later Recipe, you will design and write a full-fledged preferences system to let the user determine the default data values for a new document.

Launch Project Builder and open the project, if necessary.

1. In the source file `MySettings.m`, add the following statement at the end of the `if` block in the designated initializer, `initWithDocument:`:

```
// Default settings values
myCheckboxValue = YES;
```

   If you don't provide an initialization value for a variable, Cocoa will initialize it to `0` (or `NO`, or `nil`, or any value represented internally as `0`). For this reason, it is common practice to omit initialization of data values altogether when they should start life with one of these default values.

Here, the initialization method initialized the `myCheckboxValue` variable to `YES`. As written, this will apply both to new documents and to documents opened from storage. In the latter case, however, the value will be reset to the value read from the saved document, using routines you will shortly write in *Step 4.2*. This redundancy will be tolerated for now, since you will replace it later with a full preferences system.

Notice that you have initialized the `myCheckboxValue` directly, rather than by calling the `setMyCheckboxValue:` accessor method. This is a temporary solution to an issue you will encounter later. In *Step 5*, you will add code to the accessor method to register the action with the undo manager. You want to make sure that the initialization of the variable in this Step will *not* be registered with the undo manager, because there is no point to undoing the default value of a variable in a newly created document, so you bypass the accessor method here. Still later, however, in *Step 2* of *Recipe 2*, you will change this `initWithDocument:` method so that, instead of bypassing the `setMyCheckboxValue:` accessor method here, you will invoke it, after all, and use a better technique to avoid registering initialization of the variable with the undo manager.

Finally, notice that a `dealloc` method is not provided for MySettings. At this point, the application does not allocate any objects in MySettings, so it has nothing to release. Its super's `dealloc` method, in NSObject, will be called by the Cocoa frameworks without touching the MySetting object's layer. A `dealloc` method will become necessary in MySettings later, as you proceed to add functionality to the application.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 4: Provide for data storage and retrieval

4.3 Display the document's data

You have now initialized the document's internal data variable `myCheckboxValue` to a default value of YES, and you have written routines to set the variable from data which was read in from storage. But you have written nothing yet to enable MyWindowController to tell the window to show the value held in `myCheckboxValue` to the user, in either event.

In an application with multiple window controllers, this is the job of NSWindowController's `windowDidLoad` method, which you must override. Cocoa invokes `windowDidLoad` automatically, after the document's nib file has been loaded and all of its internals have been initialized.

Launch Project Builder and open the project, if necessary.

1. Add the following to the source file `MyWindowController.m`, before the Action methods section:

```
// Window management

- (void)windowDidLoad {
    [super windowDidLoad];
    [[self myCheckbox] setState:([[self mySettings]
myCheckboxValue] ? NSOnState : NSOffState)];
}
```

Since this is an override method, it is not necessary to declare it in the header file.

Now, whenever the window loads, whether as a new document that has never been saved or as a document that was just loaded from storage, the checkbox control in the Buttons pane of the main document window will immediately be updated to reflect the internal state of the MySettings variable `myCheckboxValue`.

You will deal later, in *Step 7*, with the case where the user chooses the Revert menu item, which does not reload the window and therefore does not cause the `windowDidLoad` method to be invoked.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 4, 2001 - 6:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 5: Implement Undo and Redo

5.2 Set the undo and redo menu item titles with localized strings

Applications that implement undo and redo should provide descriptive titles for the Undo and Redo menu items, in order to give the user a fairly specific idea of what will happen when they are chosen. This is particularly important in a Cocoa application, where multiple undo and redo are the norm. A user could easily get lost if faced with a succession of Undo menu items all titled simply "Undo".

1. In the source file `MyWindowController.m`, insert the following lines at the end of the `myCheckboxAction:` method:

```
if ([sender state] == NSOnState) {
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Checkbox", @"Name
of undo/redo menu item after checkbox control was
set")];
} else {
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Checkbox",
@"Name of undo/redo menu item after checkbox control
was cleared")];
}
```

This invocation of the undo manager's `setActionName:` method will cause the Undo or Redo

menu item's title to change whenever this user action has been invoked.

The documentation recommends that the `setActionName:` method be invoked in an action method, as you have done here, rather than in the primitive method that actually changes the data value in the model object, here, `setMyCheckboxValue:` in MySettings. A couple of reasons are given for this. For one thing, the primitive method might also be called for other purposes, and a change to the Undo and Redo menu items might not be appropriate for all of them. More importantly, the titles of the Undo and Redo menu items should reflect the nature of the user action that will be undone or redone, not the nature of the underlying primitive operation that alters the data. By changing the menu item titles in the action method, you leave open the possibility of using other menu item titles if the same change to the document were effected, say, by a menu command or an AppleScript command.

2. This is the first time you have coded for a localizable string. The use of the `NSLocalizedString()` convenience function to extract a localized string from the `Localizable.strings` resource was explained in *Step 3.1.4*. Here, you see it in use to make sure the Undo and Redo menu items appear in the local language where the application is being run (assuming the application has been localized for that language).

   To make this work, you must add the following key-value pairs to the `Localizable.strings` file in the `Resources` folder. It is customary, but not required, to name the key identically to the value used for the locale of the developer, here, English. Don't forget the trailing semicolons; if you do forget them, this will appear to work, but Cocoa will in fact be using the name of the key instead of the localized value and therefore won't pick up any different strings provided by the localization contractors.

```
/* Undo/Redo menu item names */

    /* Checkbox */
/* Name of undo/redo menu item after myCheckbox control
was set */
"Set Checkbox" = "Set Checkbox";
/* Name of undo/redo menu item after myCheckbox control
was cleared */
"Clear Checkbox" = "Clear Checkbox";
```

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 4, 2001 - 6:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 6: Review the behavior of the Save and Revert menu items

If you tested the behavior of the Undo and Redo menu items in *Step 5*, you might have noticed that the Save and Revert menu items in the File menu do not behave as a Mac OS 9 user would expect. The Save menu item in Cocoa remains enabled at all times after the first change is made to the document, even after a document is saved, reverted or opened. In Mac OS 9, the Save menu item would become disabled at these times, and it would remain disabled until the user makes another change to the document. The Revert menu item in Cocoa behaves just like Save, except that it remains disabled for a new document until the document has been saved once (because there is nothing to revert to before then). In Mac OS 9, a revert menu item is normally disabled at the same time Save is disabled.

This is the way these menu items are supposed to behave in Cocoa. The theory is that Mac OS X is inherently multi-process and multi-user in nature, and there is therefore always a chance that another application has changed the document behind the back of the current application. Because of this possibility, the Save and Revert menu items are kept available at all times, to let the application at any time update the document on disk to your current representation of it in RAM or to revert your representation in RAM to its current state on disk, in case another user, say, on a network, has altered it since you last saved your own changes. That is, it allows you to decide at any time whether to conform your version of the document to a concurrent user's version, or to force the concurrent user's version to conform to yours, or to choose Save As… to create a separate copy for your own use.

Cocoa's default behavior may leave you feeling somewhat uncomfortable. It could be considered irresponsible to allow a concurrent user to change a document's representation on disk without warning other active users about what is happening. An application could at least, as Project Builder does, raise an alert when you bring it to the front after the document has been changed out from under you by a

concurrent user, allowing you to decide what to do. Applications might also implement some form of document locking or record locking to prevent concurrent users from making changes while you are using the document. Ideally, Cocoa would implement one or more standard mechanisms for dealing with this situation. The current default, allowing a user to save or revert at any time, seems incomplete and unsatisfactory.

You will not implement record-locking or other devices here, because it is a complicated task. However, you can, if you wish, implement a simple change to the application at this point that will cause it to adopt the standard Mac OS 9 menu behavior, disabling the Save and Revert menu items unless you, yourself, have made a change to the document's representation in RAM. This may provide a small measure of safety, because it will make it slightly harder for you to overwrite changes saved to storage by others since you last saved your own changes. Note, however, that it still allows you to save your own changes.

Apple's official interface guidelines and practices should normally be followed, because Mac OS X users will come to expect all Mac OS X applications to behave alike. This is one of the Mac's great strengths. You are therefore advised not to make the changes described here. They are presented only to show you how it could be done, and to introduce you to some standard Cocoa techniques for enabling and disabling menu items.

Launch Project Builder and open the project, if necessary.

1. In the source file `MyDocument.m`, add the following method before the Persistent storage section:

```
// Menu management

- (BOOL)validateMenuItem:(NSMenuItem *)menuItem {
    if ([[menuItem title]
isEqualToString:NSLocalizedString(@"Save", @"Name of
Save menu item")]) {
        return ([self isDocumentEdited] ? YES : NO);
    } else if ([[menuItem title]
isEqualToString:NSLocalizedString(@"Revert", @"Name of
Revert menu item")]) {
        return ((([self fileName] != nil) && ([self
isDocumentEdited])) ? YES : NO);
    } else {
        return [super validateMenuItem:menuItem];
    }
}
```

   This overrides the default NSDocument implementation of `validateMenuItem:`, so you do not need to declare it in `MyDocument.h`.

Note that the override method invokes its super's method only if it is not being called on the Save or Revert menu items. Invoking super's method is necessary to ensure that Cocoa is able to validate other menu items. However, the override method prevents NSDocument's standard `validateMenuItem:` from being invoked on Save or Revert. The standard method would enable the menu item when a document exists on disk and has ever been edited, even if it has since been saved. If the standard method is not overridden, the user can choose Revert on a document that is not currently "dirtied"—that is, a document that may have been modified but which you have since saved—but there will be no response; that is, the standard revert sheet, saying that the document has been edited and asking if you want to undo the edits, does not open. This is correct behavior under the circumstances, but a user may find it confusing.

Testing whether the document's name is `nil` is a standard way to test whether it has ever been saved to storage.

You used the `NSLocalizedString()` convenience function to make sure you are comparing the menu item title to its localized name in the language where the computer is being used, as described in *Step 3.1.4*, above.

To make this work, you must add these key-value pairs to the `Localizable.strings` file in the `Resources` folder.

```
/* Name of Save menu item */
"Save" = "Save";

/* Name of Revert menu item */
"Revert" = "Revert";
```

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 7: Make the Revert menu item work

Now that you have the Save and Revert menu items highlighting according to the Mac OS 9 model or the Mac OS X model, as you prefer, you should try them out. You will notice that the Revert menu item doesn't seem to work correctly. For example, save a Vermont Recipes 1 file with its checkbox control checked. Close and then reopen that document, uncheck the checkbox, and choose File > Revert. The checkbox does not appear to revert to its checked state, as it should. However, if you use the debugger, you will discover that the value of the `myCheckboxValue` variable in MySettings has in fact returned to YES, confirming that the Revert command did in fact read the document back into memory from disk. If you trace out the logical flow of control in the source code, you will realize that there is no code in the application to tell the checkbox user control to conform its visible state to that of the `myCheckboxValue` variable when the document reverts to its saved state.

Here are the steps to follow to figure out one way to resolve this issue. They are spelled out in detail here in order to give you a real-world example of figuring out how to solve a problem in Cocoa. Later, in *Recipe 2, Step 2*, you will discover that there is an even better way.

The first step is to determine how Revert works. Checking the documentation and using the debugger to trace what happens when the Revert menu item is chosen, you discover that Cocoa automatically invokes a built-in NSDocument action method called `revertDocumentToSaved:`, which in turn invokes your override of the NSDocument `loadDataRepresentation:ofType:` method. This makes sense, because `loadDataRepresentation:ofType:` is the method that is invoked to obtain data from disk when you open a document, and you also want to obtain data from disk when you revert. Looking at the custom methods that your override of `loadDataRepresentation:ofType:` invokes, you see that you already included in one of them, `restoreFromStorage:`, a means to set the

`myCheckboxValue` variable via the MySetting model object. This is what sets the value of the document's data to match what is found on disk.

This therefore seems like a possible place to add a statement to make the checkbox's visible state match the value of the data. However, this is a very awkward place to do anything that is limited to reverting the document. The `restoreFromStorage:` method is also invoked when a document is being opened, but the `windowDidLoad` method in MyWindowController already updates the user interface in that case. To handle Revert here would require that you somehow detect whether the Open or the Revert menu item is being handled.

So you need to look a little further. You already discovered that NSDocument includes an action method for the Revert menu item, `revertDocumentToSaved:`. This may be just the ticket, since it is invoked only to revert a document, not to open a document. Still, you may have a little discomfort about overriding a Cocoa action method. As a last resort, you examine the NSDocument header file in /System/Library/Frameworks/AppKit.framework/headers/. There you hit paydirt! There is an undocumented method, `revertToSavedFromFile:ofType:`, which the comment indicates is called by the revert action method. The comment explicitly tells you that this is the appropriate place to detect when a document is being reverted and to take additional actions. Therefore, you will override this method in `MyDocument.m` and use it as a springboard to tell the window controller object that the document is reverted and the user interface needs to be updated.

1. Add the following override method to `MyDocument.m`, immediately after the `makeWindowControllers` method:

```
- (BOOL)revertToSavedFromFile:(NSString *)fileName
ofType:(NSString *)type {
    if ([super revertToSavedFromFile:fileName
ofType:type]) {
        [[self windowControllers]
makeObjectsPerformSelector:@selector(documentDidRevert)];
        return YES;
    } else {
        return NO;
    }
}
```

   This is an override method, so it does not require declaration in `MyDocument.h`.

   It first invokes the method's super, to make sure any necessary changes are made to the document's internal structure. If successful, it informs the built-in window controller array `windowControllers` that the document has reverted to its saved state by invoking the window controller's `documentDidRevert` method, which you are about to write.

1. In the header file `MyWindowController.h`, after the Accessor methods and conveniences section, add the following:

```
// Window management

- (void)documentDidRevert;
```

2. In the source file `MyWindowController.m`, after `windowDidLoad`, define the method as follows:

```
- (void)documentDidRevert {
    [self updateCheckbox:[self myCheckbox]
setting:[[self mySettings] myCheckboxValue]];
}
```

   You have seen this same statement before, in *Step 5.3*, where you used it to update the checkbox control after an undo or redo operation.

Your application can now update the user interface when a new document is created or a saved document is opened, using the `windowDidLoad` method, and when a changed document is reverted to its saved state, through the `documentDidRevert` method, both of which call a single generic method to update the checkbox. The `windowDidLoad` and `documentDidRevert` methods give you a good, general framework for handling user interface updating when a user chooses the Save, Open, and Revert menu items, as well as undo and redo, no matter how many user controls you add later.

At this point, you have achieved your goal, a working application that maintains a good separation between the model object's single data item and its single user control in the user interface. The window controller object is used to mediate between the data and the view when a change takes place in either. When the user changes the interface by clicking the control, Cocoa tells the window controller to send an action method to the model object so that it can decide how to update its data. Contrariwise, when the user changes the document's data by choosing the New, Open or Revert menu item, Cocoa tells the model object or its document to inform the window controller that it needs to decide how to update the state of the user interface.

Only two details remain to complete Recipe 1 and your first Cocoa application: create application and document icons, and revise your application's menu bar so that it discloses the name of your application to the user in appropriate menu items.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 8: 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 info window, and in various alerts and dialogs. Also, the application normally shows its icon in its About box.

This is not a tutorial on the details of using image-editing applications to create images suitable for use as icons, so you will have to create or find your own graphics to serve as icons using whatever applications are available to you. Briefly, you may find it convenient to use a drawing program, a scanner or a digital camera to acquire the images, and to use Adobe Photoshop or some other image-editing application running in Mac OS 9 or the Classic environment of Mac OS X to edit them, until native Mac OS X applications are available for the purpose.

For the Vermont Recipes icons, the cover and a page from an antique Vermont cookbook whose copyright has expired were scanned into Photoshop. Each image was then reduced in size and placed on a 128 x 128 pixel canvas. The areas outside the image were erased to transparent. Finally, each image was saved in PNG-24 format with transparency using the Save for Web… command in Photoshop 5.5. If you are a

perfectionist, you will want to repeat the process until you have saved the images in four sizes—16, 32, 48 and 128 pixels square—each optimized to look good at its size.

There is a raging controversy in some circles over whether it is best to use Apple's Mac OS X icon style—an angle view of a stylized, "photo illustrative" three-dimensional object, as described in the Icon Design chapter of Inside Mac OS X: Adopting the Aqua Interface—or a more traditional abstract, flat graphic. The Vermont Recipes choice of a flat but photo realistic icon does not reflect a considered position on the issue, but only a deficit of artistic talent. If you're serious about your application, hire a professional artist.

In this Recipe, you use icon tools provided with Mac OS X to install the icons from whatever images you have found or created. More powerful commercial applications are available, including IconBuilder Pro, Iconographer and Icon Machine.

Once you have the image files for each icon in hand, you are ready to turn them into icons for the application.

Launch IconComposer (in the */Developer/Applications/* folder). Also launch Project Builder and open the project, if necessary.

1. Using an untitled IconComposer window, drag each image onto the empty square in the first column matching its size. You can get away with using only a 128 x 128 pixel icon.

2. If each of the three smaller images is dragged onto its square, an alert may appear, asking you whether to extract a 1-bit mask from the data. Click No if the mask is already present; otherwise, click Yes.

3. Choose File > Save As..., give the icon file a name, designate any location to which to save it, and click Save. The file is automatically given the required ".icns" extension.

4. The icon is saved as an icns Browser document, so you can double-click the icon file to open it in the icns Browser application and examine it. It shows as containing only a "Thumbnail" size icon (128 x 128 pixels), but it will work fine.

5. If you did not already save them there in instruction 3., above, drag the icon files into the root level of the project folder. For Vermont Recipes, name the application icon "VRapp.icns" and the document icon "VRdoc.icns". These go in the root project folder rather than the `English.lproj` folder because icons cannot be localized.

6. Turn to Project Builder. Click the Targets tab of the project, click "Vermont Recipes 1" in the Targets pane, click the Application Settings tab, and click the Expert button.

7. On the `CFBundleIconFile` line, type **`VRapp`** without the ".icns" extension.

8. Expand the disclosure triangle for `CFBundleDocumentTypes`, then expand element 0. On the `CFBundleTypeIconFile` line, type **`VRdoc`** without the ".icns" extension.

9. Click the Files tab of the project and expand the Resources disclosure triangle. Then choose Projects > Add Files…, select the two new icon files, click Open, then click Add in the next sheet. The two new icons appear in the Groups & Files pane. Drag them into the Resources group, if necessary.

When you compile and run the application, you will see the new icons in all the expected places. (You may have to move the application from the project's build folder to the Mac OS X Applications folder, or shut down and restart, to see them in the Finder and the Dock.)

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Step 9: Revise the menu bar

Before you finish, you should perform a little cosmetic surgery on your application's menu bar.

Launch Project Builder and open the project, if necessary.

1. In the Groups & Files pane of the project window, double-click `MainMenu.nib` in the `Resources` group. `MainMenu.nib` opens in Interface Builder.
2. In the MainMenu window, double-click the New Application menu title to select its text for editing, and edit it to read **`Vermont Recipes 1`**.
3. In the MainMenu window, click once on the newly-edited Vermont Recipes 1 menu title. The menu menu opens.
4. Double-click the About NewApplication menu item to select its text for editing, and edit it to read **`About Vermont Recipes 1`**. Hit the Enter key to commit the change.
5. Double-click the Hide menu item, and edit it to read **`Hide Vermont Recipes 1`**.
6. Click the Quit menu item to select it, and edit it to read **`Quit Vermont Recipes 1`**.
7. Click the Help menu and click the NewApplication Help menu item to select it. Edit it to read **`Vermont Recipes 1 Help`**.
8. In the Interface Builder menu bar, choose File > Save.

9. Compile, link and run the application. You will find the compiled application in the `build` folder of your project folder, unless you altered your build settings in Project Builder. Try moving the application to the main `Applications` folder and running it from there, if icons don't appear properly.

Introduction > Contents > Recipe 1 > Step 9 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 1: A simple, multi-document, multi-window application

Conclusion

Run the application and try out all its features.

To ensure that Mac OS X adds your new document format to its database, don't run the application from within Project Builder the first time. Instead, open the `build` folder in the `Vermont Recipes 1` folder, move the compiled and linked application to the Mac OS X `Applications` folder, and run it by double-clicking it there. This is necessary in order to allow the system to recognize the application as the owner of its saved documents, so that double-clicking a document will automatically open it in the application, and it may also be necessary to ensure that its icons appear on the Dock and elsewhere. If you run the application from within Project Builder before running it independently, you may find that these features of Mac OS X don't work properly, and you might not be able to minimize it to the Dock.

A few things don't work yet, such as drawers, printing and help, and you haven't yet created any new menus or menu items. But an amazing number of standard application features now work perfectly, and getting to this stage required little or no effort on your part.

For example, the About Vermont Recipes menu item opens an about box with information about your application, including the copyright notice you supplied. The window minimizes and zooms as expected, both from the buttons in the window title bar and from the Window menu. Other commands in the Window menu work, for example, to bring any of multiple open windows to the front. The File > Save To… menu item creates a separate backup document on disk, leaving the front window untitled. The keyboard equivalents work just like their associated menu items. Saving changes to an existing document, or reverting it to its last saved state, opens confirmation sheets.

And all of the features you added in this first Recipe work flawlessly. Go ahead: exercise the New, Save and Open commands repeatedly, working with a dozen documents open at once, if you like. Make changes to saved documents and revert them. Use the Undo and Redo commands repeatedly to see that they count multiple changes and undo and redo them as they should.

You are ready now to turn to *Recipe 2*, where you will begin implementing a large variety of standard Mac OS X user controls. Stay tuned for still more Recipes, dealing with menus and menu items, sheets, drawers, printing, Apple Help, AppleScript, speech synthesis, and myriad other topics of interest to Cocoa developers.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe01/recipe01_conclusion.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 2: User controls—Buttons

Download the project files for Recipe 2 as a disk image and install them

Download a pdf version of Recipe 2

With this Recipe, you will begin a series of Recipes dealing with user controls. In this *Recipe 2*, you will implement a number of different kinds of buttons, which, to a Cocoa programmer, include checkboxes, radio buttons, pop-up menus, and other controls, as well as traditional push buttons. In subsequent Recipes, you will learn about other kinds of controls. For example, *Recipe 3* will cover sliders, including techniques for linking sliders and other controls. Before getting to examples of various uses for text fields in *Recipe 6*, you will digress briefly in *Recipe 4* to learn how to implement sheets, which are an important means to communicate with your users while they are using text fields, and *Recipe 5* to learn about formatters for text fields. You won't find every control known to humankind in this series, but you will find most standard Macintosh user controls, including a few variants. Some of the controls will interoperate with one another, for example, by disabling or enabling other controls or rapidly updating a number in a text box as a slider is dragged back and forth.

In these Recipes, each Step will focus on a particular control, although other, related controls may also be covered, especially where interaction among them is important. Each Step will cover both the Interface Builder and the Project Builder aspects of creating the controls it covers. After preparing the project files for *Recipe 2* in *Step 1* and improving the application's initialization code in *Step 2*, *Step 3* will provide a very detailed roadmap through the process of implementing a group of checkboxes. *Step 3* will serve as a checklist of fundamental tasks for implementing almost any kind of control. Subsequent Steps will provide less detail where repetition can easily be avoided, but new and interesting techniques will always be flagged and explained in depth. This orgnization should make it possible to use this series of Recipes as a reference when implementing controls in your own application.

The Vermont Recipes 2 application, when you have completed it, will serve as a demonstration of how the various buttons work. In subsequent Recipes in this series, user controls will be grouped by category, each in a tabbed pane of its own. Since Vermont Recipes is aimed at programmers, controls will be categorized by Cocoa view type. In some cases, this may seem odd from an end user's point of view; for example, checkboxes and pop-up menus are NSButtons, so they will appear in the Buttons tab.

There are some issues that are not covered in this series of Recipes but will be dealt with later. Chief among them is how you ensure that the items within the window are resized and repositioned appropriately when the user resizes the window. Think of the Vermont Recipes document window as having a fixed size, for now.

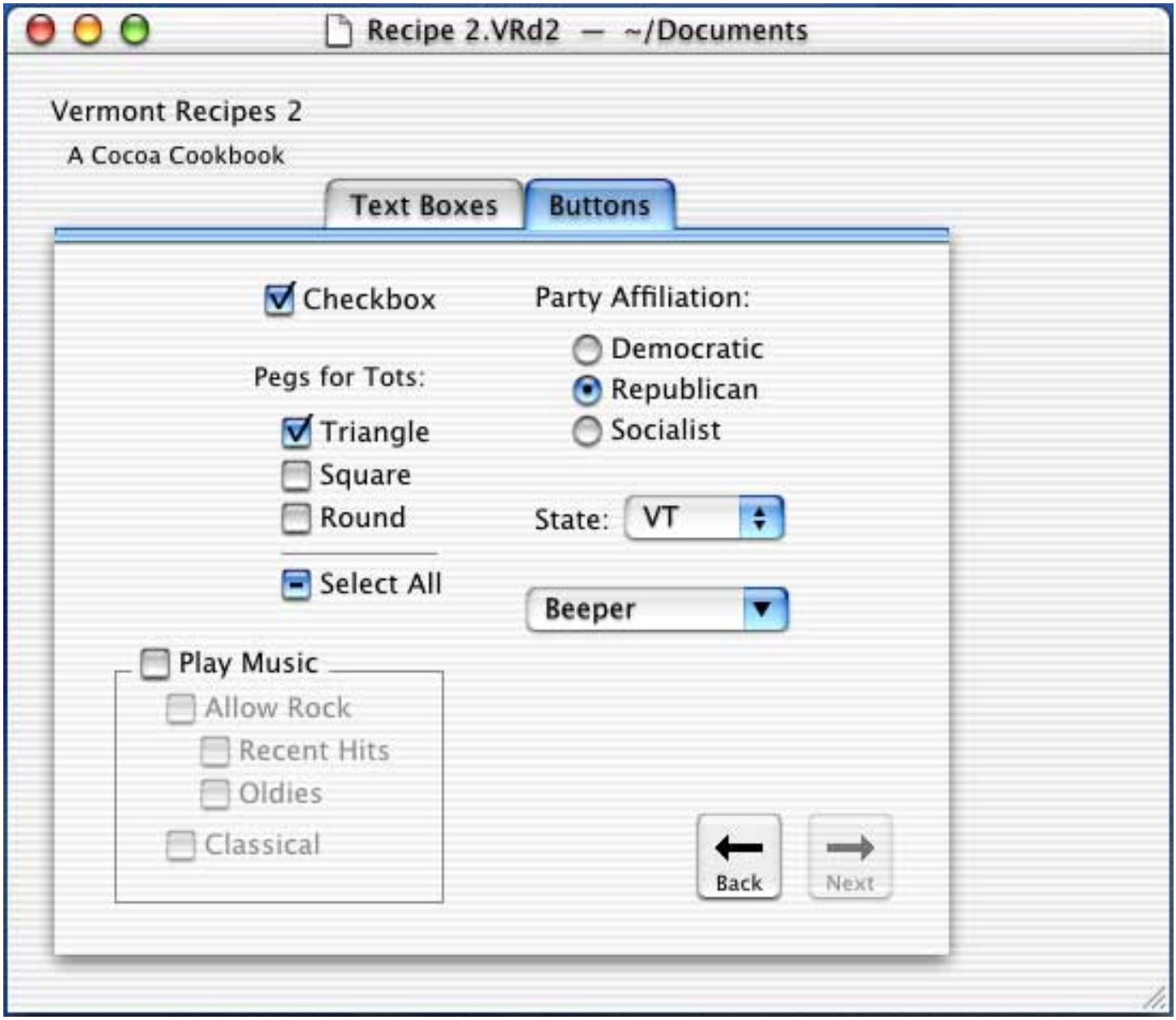
As you work your way through this series of Recipes, you should pay close attention to Mac OS X human interface guidelines. They are very detailed, prescribing specific dimensions for buttons, the position and spacing of controls in dialogs, and similar details. The success of the Mac has been built in part on the fact that users can expect things to look and work more or less the same in all applications. The prospects for Mac OS X will undoubtedly be enhanced if you conform to the official Apple standards. Interface Builder is a great help, because it lets you apply many of the guidelines automatically, particularly with the introduction of Aqua guides in Mac OS X 10.0.

Documentation

The primary source for Mac OS X human interface guidelines is Inside Mac OS X: Aqua Human Interface Guidelines, a downloadable pdf document. Substantial additions and many changes have appeared over the last several months, and more are expected. Professional developers will want to keep close tabs on the status of this document.

The Mac OS X guidelines are written as addenda to the older human interface guideline documents, which remain in effect to the extent they are not inconsistent with the Mac OS X document. The older documents are available on the web as the downloadable pdf files Macintosh Human Interface Guidelines and Mac OS 8 Human Interface Guidelines. Both are also available as browsable web documents.

Screenshot 2-1: The Vermont Recipes 2 application



Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe02/recipe02.html

Introduction > Contents > Recipe 2 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 1: Prepare the project for Recipe 2

Before getting into the meat of Recipe 2, you should change some of the application's settings to reflect the fact that you are creating a new version of the application. Instead of changing the application's name to "Vermont Recipes 2" for this Recipe, which would require changing it again for every subsequent Recipe, it will be more efficient simply to give it its final name of "Vermont Recipes". To reflect the fact that it will incorporate new material added in this *Recipe 2*, you will simply bump up the application's development version. Also, the new data items will require a new file format, so you will change the document type codes, to make sure that the Finder does not try to open old documents left over from *Recipe 1* in the new application. The change in the application's name will also require changing some resources and one of the source files. In subsequent Recipes, you will only have to change the application signature and some document type codes.

1. In the Finder, duplicate the `Vermont Recipes 1` folder and name the copy **`Vermont Recipes 2`**. It is a good idea to give the top-level project folders different names to reflect different builds, to make it easy to identify them in the Finder and to save backups or archives. Put the `Vermont Recipes 1` folder away in a safe place, or discard it.

2. In the `Vermont Recipes 2` folder, rename the `Vermont Recipes 1.pbproj` bundle as **`Vermont Recipes.pbproj`**. There is no need to change this name every time you build a new version. Even with multiple versions open at once in Project Builder, you will be able to tell them apart by the path available by Command-clicking on the project name in the title bar of the project window.

3. Drag the entire `build` folder to the trash. Cocoa will create a new one for you the next time you build the application.

4. Double-click `Vermont Recipes.pbproj` to open the project in Project Builder.

5. In the project window, click the Targets tab and select `Vermont Recipes 1`. Choose Project > Rename, then rename the target to **`Vermont Recipes`**.

6. Click the Build Settings tab and change the Product name to **`Vermont Recipes`**.

7. Click the Application Settings tab and the Expert button, and change several settings, as follows:

    a. Expand `CFBundleDocumentTypes` and element 0, then expand `CFBundleTypeExtensions`, and change the setting of element 0 from "`VRd1`" to **`VRd2`**.

    b. Do the same with `CFBundleTypeOSTypes`.

    c. Change `CFBundleTypeName` to **`Vermont Recipes Document Format`**.

    d. Change `CFBundleExecutables` to **`Vermont Recipes`**, if it hasn't already changed.

    e. Change `CFBundleIdentifier` to **`com.stepwise.VermontRecipes.VermontRecipes`**.

    f. Change `CFBundleSignature` to **`VRa2`**.

    g. Change `CFBundleVersion` to **`1.0.0d2`**.

8. Click the Files tab, expand the `Resources` group, and click `InfoPlist.strings`. Change every instance of "`Vermont Recipes 1`" to **`Vermont Recipes`**, and change every instance of "`1.0.0d1`" to **`1.0.0d2`**.

9. Double-click `MyDocument.nib` to open it in Interface Builder. At the top of the main document window, change "`Vermont Recipes 1`" to **`Vermont Recipes`**. This will require you to recenter the line below, "A Cocoa Cookbook", and to move both lines to the left until they again comply with Aqua Human Interface Guidelines for the distance of items from the left edge of a window.

10. Back in the Project Builder window, double-click `MainMenu.nib` to open it in Interface Builder. Click on the Apple and Help menus and change every instance of "`Vermont Recipes 1`" to **`Vermont Recipes`**.

11. In Project Builder, expand the Classes topic, and open `MyDocument.m`. A little over half way down, in the Keys and values for dictionary section, change myDocumentType to **`@"Vermont Recipes Document Format"`**.

    While you're at it, change `currentMyDocumentVersion` to **`2`**. In real world development, this

would be used by backward-compatibility routines to help identify the format of the document when opening it.

12. In the `Localizable.strings` file and all of the header and source files, change the application name in any comments you may have placed at the top to **`Vermont Recipes`**.

13. Build and run the application in Project Builder to confirm that it works as expected.

You are now ready to begin adding user controls to the application.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 2: A better way to handle data initialization

In *Recipe 1*, in the course of laying out the basic features of the Vermont Recipes application, you implemented a single user control, a checkbox or "switch button." This required you to take care of a number of details needed to implement any user control. Before you begin implementing additional controls in *Recipe 2* and subsequent Recipes, you should revisit *Recipe 1* to ensure that you are using the best possible techniques to handle the checkbox. It turns out that there are a few things you can do better.

In this *Step 2*, you will improve your data initialization code, and this will permit you to make further improvements to the way the checkbox is updated on screen when a new document is created, when an existing document is opened, and when a document is reverted to its saved state. In doing this, you are following in our footsteps on a beginner's journey toward understanding Cocoa. You should always remain alert to overlooked opportunities, and be prepared to go back and improve code already written if advantages are apparent. One of the most important benefits to consider is that your code will become easier to maintain and upgrade in the future.

1. In *Recipe 1, Step 4.1*, you took care to avoid registering with the undo manager when you initialized the data variable associated with the checkbox. You did this by taking a direct and expedient route: you simply bypassed the data variable's `setMyCheckbox:` accessor method, where registration with the undo manager is handled, and instead set the data variable directly. However, this violates a strong preference in Cocoa programming for accessing data variables only through accessor methods.

   It turns out that Cocoa provides an easy way to suppress registration with the undo manager: the `disableUndoRegistration` and `enableUndoRegistration` methods declared in

NSUndoManager. If you bracket a call to your data variable's `setMyCheckbox:` accessor method with invocations of these two NSUndoManager methods, the accessor method will not register the change with the undo manager. You will now use this technique in the initialization of the data variable associated with the checkbox control when a new document is created.

In `MySettings.m`, replace the line of the `initWithDocument` method that initializes the Checkbox data value, `myCheckboxValue = YES;`, with the following statements:

```
[[self undoManager] disableUndoRegistration];
[self setMyCheckboxValue:YES];
[[self undoManager] enableUndoRegistration];
```

Later, in this and subsequent Recipes, when you initialize additional data variables, you will always place the calls to their accessor methods between these two NSUndoManager methods.

2. In *Recipe 1, Step 4.2*, you also took care to avoid registering with the undo manager when you loaded the checkbox value from disk upon opening an existing document or reverting a document to its saved state, which is a form of initialization that also requires bypassing undo manager registration. The cure is identical.

   In `MySettings.m`, replace the contents of the `restoreFromDictionary:` method with the following:

   ```
   [[self undoManager] disableUndoRegistration];
   [self setMyCheckboxValue:(BOOL)[[dictionary
   objectForKey:myCheckboxValueKey] intValue]];
   [[self undoManager] enableUndoRegistration];
   ```

   Later, in this and subsequent Recipes, you will add additional calls to accessor methods between the two NSUndoManager methods.

3. Now, a remarkable thing happens. By using the accessor method to initialize the data variable in the `restoreFromDictionary:` method, it becomes possible to update the checkbox when a document is reverted to its saved state by taking advantage of the fact that the accessor method posts a notification. The specific view updater method you wrote in *Recipe 1, Step 5.3* is a notification method that is registered to receive and act upon this notification. The checkbox control will therefore update automatically on screen when it receives the notification, not only when the user invokes Undo or Redo, but also when the user reverts a document to its saved state. You no longer need to invoke the generic updater method explicitly in MyWindowController's `documentDidRevert` method. In fact, you no longer need the `documentDidRevert` method at all, nor the `revertToSavedFromFile:ofType:` override method that you added to `MyDocument.m` in *Recipe 1, Step 7*. You are able to simplify your code by removing these two

methods.

Remove the `documentDidRevert` method declaration and definition from both `MyWindowController.h` and `MyWindowController.m`. Leave the "// Window management" heading in place in `MyWindowController.h` even though it will be left empty, as you will add a method to it in a moment.

In `MyDocument.m`, remove the definition of the `revertToSavedFromFile:ofType:` override method.

4. Unfortunately, you still need to update the checkbox control explicitly in MyWindowController's `windowDidLoad` method. The reason for this is that, when you initialize the data variable in the MySettings object's initialization method or its load data method when a document is being created or opened, its accessor method posts a notification before the control's update method in the MyWindowController object has been registered as an observer. The checkbox user control doesn't yet even exist, because a document that is being created or opened is initialized before its window appears or its window controller object is instantiated. The posting of the notification when nobody is yet observing it is harmless, but you do have to update the user control yourself later, when the window and its controller are created.

   There is a better way to do even this, however. When you originally set up the checkbox updater routines, you created both a specific updater for this user control and a generic updater for all two-state checkboxes. For reasons that do not now seem important, you invoked the generic updater in the `windowDidLoad` method in *Recipe 1, Step 5.3*. It will make for greater ease of maintenance and upgrading if you invoke the specific updater, instead, because specific updaters for new controls may want to do special things that cannot be implemented in a generic updater for all controls of that kind. Although the specific updater is a notification method, you can invoke it directly and pass `nil` in its notification parameter.

   While you're at it, because you know you will shortly add a large number of controls to the application, it might be convenient to add a method where all calls to view updater methods can be collected in one place. You can call it `updateWindow` and place all of the invocations of view update methods into it. Place the new method at the end of the `Window management` section, both in `MyWindowController.h` and in `MyWindowController.m`., and invoke it once at the end of the `windowDidLoad` method.

   In `MyWindowController.h`, declare the new `updateWindow` method in the recently emptied Window management section, as follows:

```
- (void)updateWindow;
```

   In `MyWindowController.m`, define it as follows after the `windowDidLoad` method:

```
- (void)updateWindow {
    [self updateMyCheckbox:nil];
}
```

In the `windowDidLoad` method definition in `MyWindowController.m`, replace the second line, after `[super windowDidLoad]`, with an invocation of the new `updateWindow` method, as follows:

```
[self updateWindow];
```

5. As long as you're cleaning up the file, you might as well also create a separate method to hold all the notification observer registrations that you will have to create as you add more user controls.

   In `MyWindowController.h`, at the top of the Window management section before the `updateWindow` method, declare a new `registerNotificationObservers` method:

   ```
   - (void)registerNotificationObservers;
   ```

   In `MyWindowController.m`, define it as follows, after `windowDidLoad`:

   ```
   - (void)registerNotificationObservers {
       [[NSNotificationCenter
   defaultCenter]addObserver:self
   selector:@selector(updateMyCheckbox:)
   name:VRMyCheckboxValueChangedNotification object:[self
   mySettings]];
   }
   ```

   In the `windowDidLoad` method definition in `MyWindowController.m`, replace the notification observer registration statement with an invocation of the new `registerNotificationObservers` method immediately before the call to `[self updateWindow]`, as follows:

   ```
   [self registerNotificationObservers];
   ```

You have now prepared a solid foundation for implementing a number of new user controls.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 7, 2001 - 6:00 AM

Recipe 2: User controls—Buttons

Step 4: Checkboxes (switch buttons) in a bordered group box

- Highlights:
  - Using a control as a group box title
  - Disabling and enabling controls

In this Step, you will implement five more standard two-state checkboxes, or switch buttons. This Step differs from the previous Step in three respects: First, the topmost checkbox serves as the title of the group. Second, it enables and disables the rest of the controls in the group. Another checkbox enables and disables a subgroup. Finally, the group is surrounded by a box with borders because this makes its functioning clearer to the user. Boxes are permitted in these circumstances by the Aqua Human Interface Guidelines.

The routine code in this Step is very similar—nearly identical—to much of the code in the previous Step. The explanations will therefore be kept to a minimum, simply showing the code to be inserted. When in doubt about the reason for any code snippet, refer to the corresponding instruction in *Step 3*.

The routine steps needed to implement these controls are covered in instruction 2., below. The interesting material is covered in instruction 1 and in instructions 3. and following. Here, the interesting material has to do with turning a control into a group box title in Interface Builder and using Project Builder to code the enabling and disabling of the remaining user controls.

1. Use Interface Builder to create several new checkboxes grouped together. When you are done with this instruction 1, the group should look like *Screenshot 2-3*.

a. Open `MyDocument.nib` in Interface Builder and select the Buttons tab view item, if necessary. Make sure the tab view item is selected, so that controls dropped in from the palette land on that pane, not in the underlying tab view.

b. From the Views palette, drag five checkboxes onto the Buttons pane and arrange them approximately as shown in Screenshot 2-3.

c. Rename the new checkboxes, from top to bottom, **`Play Music`** (without a trailing colon), **`Allow Rock`**, **`Recent Hits`**, **`Oldies`**, and **`Classical`**.

d. Interface Builder doesn't let you turn the built-in title of a group box into a user control, although the Aqua Human Interface Guidelines expressly allow the use of a checkbox label or the text of a pop-up menu as a group box title, in addition to static text. Presumably, Interface Builder will eventually be updated to make this possible directly, but in the meantime you must devise a technique of your own.

   Your first thought on creating a title consisting of a checkbox might be to use a group box without a title and simply drag the control into the place where the title would normally appear. Unfortunately, if you do that, the top border of the box will show through.

   You must therefore resort to a kludge to accomplish the desired effect. Select the last four new switches, then choose Layout > Group in Box. A box appears surrounding the selected items, with a title at the top having the default title "Title." Drag the fifth checkbox, Play Music, until it covers the Title, then line them up on the same text baseline and align their left edges; use the arrow keys as necessary to nudge the checkbox into position. Then choose Layout > Send to Back. Select the title for editing, delete the word "Title," and type enough space characters to expand the text area horizontally so that it is a couple of pixels wider than the Play Music checkbox. You will be left with a top border that has a gap just wide enough to accept the Play Music checkbox. Choose Layout > Send to Back immediately while the box is still selected.

   There were some old NextStep developer's tricks to make the checkbox opaque, so that you wouldn't need a dummy title composed of spaces. One was to group the control in a borderless, untitled box, then drag it over the top border. However, in Mac OS X this technique no longer works, because the checkbox is clipped to invisibility as you drag it away from the box. Another technique still works. You can create a non-editable, non-selectable, non-bordered text box with no content but a suitable background color to obscure the top border of the group box. The dummy title with spaces is easier to implement, however, so stick with it.

e. You don't have to reposition any of the checkboxes within the box to comply with Aqua guides, because the Group in Box command took care of that for you.

f. There is one final issue posed by the simulated group box title, namely, moving the box and its checkbox title into position in the tab view item. If you drag the box without also selecting

the simulated title, their positioning relative to one another will be disrupted. The only real solution is to remember to select both of them before dragging. It used to be possible to make them draggable together without having to select both, by grouping the box and the checkbox into another box, but this no longer seems to provide foolproof protection against accidentally dragging the inner box out of position relative to the checkbox that serves as a title. For now, select both and drag them into position. The left border of the group should be placed the same distance from the left edge of the tab view item as the Checkbox control, and the vertical gap between this group and the group above it should be the same as the gap between that group and the Checkbox control above it.

2. Use Project Builder to write the code.

    a. **User control outlet variable and accessors**. In the header file `MyWindowController.h`, declare five new outlets to access each of the new checkboxes, after the Pegs switch button group, as follows:

```
// Music switch button group
IBOutlet NSButton *musicCheckbox;
IBOutlet NSButton *rockCheckbox;
IBOutlet NSButton *recentRockCheckbox;
IBOutlet NSButton *oldiesRockCheckbox;
IBOutlet NSButton *classicalCheckbox;
```

    Still in the header file, also declare accessors for the outlets after the Pegs switch button group in the Accessor methods and conveniences section, as follows:

```
// Music switch button group
- (NSButton *)musicCheckbox;
- (NSButton *)rockCheckbox;
- (NSButton *)recentRockCheckbox;
- (NSButton *)oldiesRockCheckbox;
- (NSButton *)classicalCheckbox;
```

    In the source file `MyWindowController.m`, define the accessors after the Pegs switch button group in the Accessor methods and conveniences section, as follows:

```
// Music switch button group

- (NSButton *)musicCheckbox {
    return musicCheckbox;
}

- (NSButton *)rockCheckbox {
    return rockCheckbox;
}

- (NSButton *)recentRockCheckbox {
    return recentRockCheckbox;
}

- (NSButton *)oldiesRockCheckbox {
    return oldiesRockCheckbox;
}

- (NSButton *)classicalCheckbox {
    return classicalCheckbox; }
```

b. **Data variable and accessors**. All of the new checkboxes require corresponding Boolean variables in MySettings to hold the data they represent. The first checkbox ("Play Music:") is used to determine whether music is turned on or off, so this value must be preserved. The second switch ("Allow Rock") is used to control whether playing of rock music is allowed, so it also needs a data variable. The remaining three switches record what kinds of music have been selected. Two of them are subcategories of rock.

   In the header file `MySettings.h`, declare four new variables after the Pegs section, as follows:

```
// Music
BOOL musicValue;
BOOL rockValue;
BOOL recentRockValue;
BOOL oldiesRockValue;
BOOL classicalValue;
```

   In `MySettings.h`, also declare the corresponding accessor methods after the Pegs subsection of the Accessor methods and conveniences section, as follows:

```
// Music

- (void)setMusicValue:(BOOL)value;
- (BOOL)musicValue;

- (void)setRockValue:(BOOL)value;
- (BOOL)rockValue;

- (void)setRecentRockValue:(BOOL)value;
- (BOOL)recentRockValue;

- (void)setOldiesRockValue:(BOOL)value;
- (BOOL)oldiesRockValue;

- (void)setClassicalValue:(BOOL)value;
- (BOOL)classicalValue;
```

Turn to the source file `MySettings.m` and define the accessor methods after the `Pegs` subsection, as follows:

```
// Music

- (void)setMusicValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setMusicValue:musicValue];
    musicValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRMusicValueChangedNotification
object:self];
}

- (BOOL)musicValue {
    return musicValue;
}

- (void)setRockValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setRockValue:rockValue];
    rockValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRRockValueChangedNotification
object:self];
```

```
}

- (BOOL)rockValue {
    return rockValue;
}

- (void)setRecentRockValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setRecentRockValue:recentRockValue];
    recentRockValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRRecentRockValueChangedNotification
object:self];
}

- (BOOL)recentRockValue {
    return recentRockValue;
}

- (void)setOldiesRockValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setOldiesRockValue:oldiesRockValue];
    oldiesRockValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VROldiesRockValueChangedNotification
object:self];
}

- (BOOL)oldiesRockValue {
    return oldiesRockValue;
}

- (void)setClassicalValue:(BOOL)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setClassicalValue:classicalValue];
    classicalValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRClassicalValueChangedNotification
object:self];
}

- (BOOL)classicalValue {
    return classicalValue;
```

```
}
```

c. **Notification variable**. Return to the header file `MySettings.h`, at the bottom of the file, to declare the notification variables used in the `set`… methods, as follows:

```
// Music
extern NSString *VRMusicValueChangedNotification;
extern NSString *VRRockValueChangedNotification;
extern NSString
*VRRecentRockValueChangedNotification;
extern NSString
*VROldiesRockValueChangedNotification;
extern NSString
*VRClassicalValueChangedNotification;
```

Turn back to the source file `MySettings.m`, near the top of the file, to define the notification variables, as follows:

```
// Music
NSString *VRMusicValueChangedNotification =
@"MusicValue Changed Notification";
NSString *VRRockValueChangedNotification =
@"RockValue Changed Notification";
NSString *VRRecentRockValueChangedNotification =
@"RecentRockValue Changed Notification";
NSString *VROldiesRockValueChangedNotification =
@"OldiesRockValue Changed Notification";
NSString *VRClassicalValueChangedNotification =
@"ClassicalValue Changed Notification";
```

d. **GUI update method**. Go now to the header file `MyWindowController.h` to declare methods to update the graphical user interface in response to these notifications, after the Pegs subsection of the Specific view updaters section, as follows:

```
// Music
- (void)updateRecentRockCheckbox:(NSNotification
*)notification;
- (void)updateOldiesRockCheckbox:(NSNotification
*)notification;
- (void)updateClassicalCheckbox:(NSNotification
*)notification;
```

You will also need update methods for the Play Music and Rock checkboxes, but this involves some new thinking and will be deferred to instruction 4., below.

In the source file `MyWindowController.m`, define these specific update methods, after the Pegs subsection of the Specific view updaters section, as follows:

```
// Music

- (void)updateRecentRockCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self recentRockCheckbox]
setting:[[self mySettings] recentRockValue]];
}

- (void)updateOldiesRockCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self oldiesRockCheckbox]
setting:[[self mySettings] oldiesRockValue]];
}

- (void)updateClassicalCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self classicalCheckbox]
setting:[[self mySettings] classicalValue]];
}
```

e. **Notification observer**. Now register the window controller as an observer of the notifications that will trigger these updaters, by inserting the following statements in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the Pegs registrations:

```
// Music
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateRecentRockCheckbox:)
name:VRRecentRockValueChangedNotification
object:[self mySettings]];
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateOldiesRockCheckbox:)
name:VROldiesRockValueChangedNotification
object:[self mySettings]];
[[NSNotificationCenter defaultCenter]
```

```
addObserver:self
selector:@selector(updateClassicalCheckbox:)
name:VRClassicalValueChangedNotification
object:[self mySettings]];
```

The Play Music and Rock switch updaters will be dealt with in instruction 5., below.

f. **Action method**. Next, you must add action methods. In the header file `MyWindowController.h`, after the Pegs section at the end, add the following:

```
// Music
- (IBAction)recentRockAction:(id)sender;
- (IBAction)oldiesRockAction:(id)sender;
- (IBAction)classicalAction:(id)sender;
```

You will also need action methods for the Play Music and Rock switches, but these will be deferred to instruction 3., below, because both require some new techniques.

In the source file `MyWindowController.m`, define these action methods, after the Pegs section at the end,, as follows:

```
// Music

- (IBAction)recentRockAction:(id)sender {
    [[self mySettings] setRecentRockValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Recent Hits",
@"Name of undo/redo menu item after Recent Hits
checkbox control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Recent
Hits", @"Name of undo/redo menu item after Recent
Hits checkbox control was cleared")];
    }
}

- (IBAction)oldiesRockAction:(id)sender {
    [[self mySettings] setOldiesRockValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
```

```
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Oldies",
@"Name of undo/redo menu item after Oldies checkbox
control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Oldies",
@"Name of undo/redo menu item after Oldies checkbox
control was cleared")];
    }
}

- (IBAction)classicalAction:(id)sender {
    [[self mySettings] setClassicalValue:([sender
state] == NSOnState)];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Classical",
@"Name of undo/redo menu item after Classical
checkbox control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Classical",
@"Name of undo/redo menu item after Classical
checkbox control was cleared")];
    }
}
```

g. **Localizable.strings**. Don't forget to update the `Localizable.strings` file with the undo and redo menu titles in the previous instruction.

h. **Initialization**. Don't initialize any of your new variables. When a new document is opened, the Play Music checkbox will be unchecked and all of the other switches will be unchecked and disabled.

i. **Data storage**. Take care now of persistent storage of the new data variables. In the source file `MySettings.m`, define these keys at the end of the Keys and values for dictionary subsection of the Persistent storage section:

```
// Music
static NSString *musicValueKey = @"MusicValue";
static NSString *rockValueKey = @"RockValue";
static NSString *recentRockValueKey =
@"RecentRockValue";
static NSString *oldiesRockValueKey =
@"OldiesRockValue";
static NSString *classicalValueKey =
@"ClassicalValue";
```

Add these lines at the end of the `convertToDictionary:` method, using the new methods from the VRStringUtilities category that you created in *Step 3*:

```
// Music
[dictionary setObject:[NSString stringWithBool:[self
musicValue]] forKey:musicValueKey];
[dictionary setObject:[NSString stringWithBool:[self
rockValue]] forKey:rockValueKey];
[dictionary setObject:[NSString stringWithBool:[self
recentRockValue]] forKey:recentRockValueKey];
[dictionary setObject:[NSString stringWithBool:[self
oldiesRockValue]] forKey:oldiesRockValueKey];
[dictionary setObject:[NSString stringWithBool:[self
classicalValue]] forKey:classicalValueKey];
```

Add these lines near the end of the `restoreFromDictionary:` method, before the `[[self undoManager] enableUndoRegistration]` statement, also using the new VRStringUtilities methods:

```
// Music
[self setMusicValue:[[dictionary
objectForKey:musicValueKey] boolValue]];
[self setRockValue:[[dictionary
objectForKey:rockValueKey] boolValue]];
[self setRecentRockValue:[[dictionary
objectForKey:recentRockValueKey] boolValue]];
[self setOldiesRockValue:[[dictionary
objectForKey:oldiesRockValueKey] boolValue]];
[self setClassicalValue:[[dictionary
objectForKey:classicalValueKey] boolValue]];
```

j. **GUI update method invocation**. Finally, add calls to the control update methods to the

window controller. In `MyWindowController.m`, add the following calls at the end of the `updateWindow` method:

```
// Music
[self updateRecentRockCheckbox:nil];
[self updateOldiesRockCheckbox:nil];
[self updateClassicalCheckbox:nil];
```

You will deal with the Play Music and Rock switches in instruction 5., below.

3. You have left until the end the most interesting part of this Step, implementing the means to disable and enable various groupings of checkboxes.

    a. You will start with the two subsidiary checkboxes under the Rock checkbox, the Recent Hits and Oldies checkboxes.

    First, go back to Interface Builder. Select in turn the Recent Hits and the Oldies checkboxes. In the NSButton Info palette, under Options, deselect the Enabled checkbox for each, so that they will start up in a default disabled state.

    Then, in the header file `MyWindowController.h`, declare a new action method as follows, at the top of the Music subsection in the Action methods and conveniences section:

```
- (IBAction)rockAction:(id)sender;
```

    Then turn to the definition of this action method, which disables and enables the two subsidiary checkboxes, the Recent Hits and Oldies checkboxes. In the source file `MyWindowController.m`, define its action method at the top of the Music subsection in the Action methods and conveniences section, as follows:

```
- (IBAction)rockAction:(id)sender {
    [[self mySettings] setRockValue:([sender state]
== NSOnState)];
    [[self recentRockCheckbox] setEnabled:[[self
mySettings] rockValue]];
    [[self oldiesRockCheckbox] setEnabled:[[self
mySettings] rockValue]];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Allow Rock",
@"Name of undo/redo menu item after Allow Rock
checkbox control was set")];
    } else {
```

```
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Allow Rock",
@"Name of undo/redo menu item after Allow Rock
checkbox control was cleared")];
    }
}
```

You could have tested the on or off state of the Allow Rock checkbox to determine whether to enable or disable the two subsidiary checkboxes, instead of testing the state of the `rockValue` variable. It is generally safer, however, to be consistent about relying on a data variable whenever one exists, in order to minimize the possibility of the application's data and its user interface getting out of sync.

Don't forget to provide for the localizable Undo/Redo menu item names in `Localizable.strings`.

b. The action method for the Play Music checkbox is more complicated. While considering whether to enable and disable the subsidiary checkboxes in the group, it must take into account the enabled or disabled state of the Rock checkbox with respect to its own sub-subsidiary switches.

It will take several lines of code to enable or disable all of the subsidiary checkboxes in the Play Music group. Thinking ahead, you anticipate that these same lines of code will have to be called both in the action method and in the update method, since both methods must enable or disable the group based on identical considerations. After all, a user might change the Play Music setting variously by clicking the checkbox, by sending an AppleScript command, by choosing Undo or Redo, or by reverting the document to its saved state. For efficiency's sake, therefore, you first create a utility method to accomplish this task.

In the header file `MyWindowController.h`, add the following declaration at the top of the Music subsection of the Specific view updaters section:

```
- (void)enableMusicGroup:(BOOL)flag;
```

In the source file `MyWindowController.m`, define the utility method at the top of the Music subsection of the Specific view updaters section:

```
- (void)enableMusicGroup:(BOOL)flag {
    [[self rockCheckbox] setEnabled:flag];
    [[self recentRockCheckbox] setEnabled:flag &&
[[self mySettings] rockValue]];
    [[self oldiesRockCheckbox] setEnabled:flag &&
[[self mySettings] rockValue]];
    [[self classicalCheckbox] setEnabled:flag];
}
```

As you see, the subsidiary Recent Hits and Oldies checkboxes are updated on the basis both of the Play Music checkbox and the Allow Rock checkbox. If the Allow Rock checkbox is off, for example, then setting the Play Music checkbox should not enable the subsidiary Rock catagories.

Now you can turn to the `musicAction:` method. In the header file `MyWindowController.h`, declare a new action method as follows, at the top of the Music subsection in the Action methods section:

```
- (IBAction)musicAction:(id)sender;
```

At the top of the Music subsection of the Action methods section of the source file `MyWindowController.m`:

```
- (IBAction)musicAction:(id)sender {
    [[self mySettings] setMusicValue:([sender state]
== NSOnState)];
    [self enableMusicGroup:[[self mySettings]
musicValue]];
    if ([sender state] == NSOnState) {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Play Music",
@"Name of undo/redo menu item after Play Music
checkbox control was set")];
    } else {
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Clear Play Music",
@"Name of undo/redo menu item after Play Music
checkbox control was cleared")];
    }
}
```

The default state of the Play Music checkbox will be off, so return to Interface Builder and

uncheck the Enabled checkbox in the Options section of the NSButton Info palette for the Allow Rock and Classical checkboxes.

Finally, make sure the localizable Undo/Redo menu item names have been provided for in `Localizable.strings`.

4. Now for the update methods.

    a. The implementation of the Play Music update method is very simple, because of the utility method you created in instruction 3.b., above. In addition to doing what an update method normally does—setting the state of its checkbox to on or off—it need only invoke the utility method to enable or disable all of the other checkboxes in the group.

    In the header file `MyWindowController.h`, declare the update method as follows, at the top of the Music subsection of the Specific view updaters section:

```
- (void)updateMusicCheckbox:(NSNotification
*)notification;
```

    In the source file `MyWindowController.m`, define the update method as follows, at the top of the Music subsection of the Specific view updaters section:

```
- (void)updateMusicCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self musicCheckbox]
setting:[[self mySettings] musicValue]];
    [self enableMusicGroup:[[self mySettings]
musicValue]];
}
```

    Now make sure the `updateMusicCheckbox:` method gets called when its data value is changed. In the source file `MyWindowController.m`, add the following to the `registerNotificationObservers` method, at the beginning of the Music section:

```
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateMusicCheckbox:)
name:VRMusicValueChangedNotification object:[self
mySettings]];
```

    b. Do the same with the Allow Rock switch.

In the header file `MyWindowController.h`, declare the update method as follows, after the `enableMusicGroup:` method:

```
- (void)updateRockCheckbox:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define the update method as follows, after the `enableMusicGroup:` method:

```
- (void)updateRockCheckbox:(NSNotification
*)notification {
    [self updateCheckbox:[self rockCheckbox]
setting:[[self mySettings] rockValue]];
    [[self recentRockCheckbox] setEnabled:[[self
mySettings] rockValue] && [[self mySettings]
musicValue]];
    [[self oldiesRockCheckbox] setEnabled:[[self
mySettings] rockValue] && [[self mySettings]
musicValue]];
}
```

You might initially think that the `updateRockCheckbox:` method doesn't need to test the status of the Play Rock checkbox when updating the two subsidiary Rock checkboxes, because if the Play Rock checkbox is unchecked the Rock checkbox will be disabled and the user can't select it to invoke this method. However, you must always keep in mind that there are other interfaces, such as AppleScript, that might be able to do things without paying attention to GUI constraints (even if that might be a bug). Code defensively, and test the state of all conditions on which updating a user control depends.

Now make sure the `updateRockCheckbox:` method gets called when its data value is changed. In the source file `MyWindowController.m`, add the following to the `registerNotificationObservers` method, after the `VRMusicValueChangedNotification` registration:

```
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateRockCheckbox:)
name:VRRockValueChangedNotification
object:[self mySettings]];
```

5. One thing remains. The graphical user interface must be updated when a document is created or

opened . As noted in instruction 2.j., above, you must therefore update the visible state of the Play Music and Rock checkboxes. In `MyWindowController.m`, add the following calls at the beginning of the Music section in the `updateWindow` method:

```
[self updateMusicCheckbox:nil];
[self updateRockCheckbox:nil];
```

Also, the `enableMusicGroup:` method must be invoked in the `windowDidLoad` method:

```
[self enableMusicGroup:[[self mySettings] musicValue]];
```

Notice that the `enableMusicGroup:` method does double duty here, enabling and disabling both the Allow Rock subgroup and the Play Music group.

6. Before you can run the revised application, you must inform the nib file of the new outlets and actions you have created in the code files, then connect them to the new checkboxes.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two Vermont Recipes 2 header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

    b. Select the Instances tab and Control-drag from the File's Owner icon to each of the five new checkboxes in turn and, each time, click its outlet name then click the Connect button in the Connections pane of the File's Owner Info palette.

    c. Control drag from each of the five new checkboxes to the File's Owner icon in turn and, each time, click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSButton Info palette, then click the Connect button.

7. Compile and run the application to test the interactions among the five new checkboxes. Be sure to explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly.

    Take this opportunity to reconsider the user interface. You decided early in this Step that it was advisable to make this set of controls a boxed group with a border. But is the border really necessary? To explore this question, go into Interface Builder, select the NSBox object by clicking in an empty area near one of the checkboxes, and, in the NSBox Info palette, click the borderless Box Type button. Save, build and run the application. It may be a close question, but a group with a control for its title and subgroups within the main group, where the title disables all the controls in the group, seems clearer with a border. Why don't you set it back to a bordered box and move on to the next Step.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe02/recipe02_step04.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 5: A radio button cluster

- Highlights:
  - Using tags and an enumeration type to manage a radio button cluster

Radio buttons always come in clusters. A two-button cluster is sometimes preferred over a checkbox because it allows you to name the two states; for example, a choice between "red" and "green" is more meaningful than the choice whether to turn "red" on or off. More than two radio buttons in a cluster are very common.

Interestingly, no matter how many radio buttons are in a cluster, only one variable is needed to store their value. This follows from the fact that only one button in the cluster can be selected at a time. The value of the cluster can therefore be specified by an integer representing the ordinal index of the selected button, starting with 0 for the first. It is customary, though not necessary, to use the C enumeration type in order to assign meaningful names to each of the integer values.

Cocoa implements radio clusters and certain other grouped controls using matrices whose component cells can be assigned "tags". Each button in a radio button cluster is a separate cell in an NSMatrix object. You can use the NSMatrix Info palette in Interface Builder to assign each button a unique tag value, which you can use in your code to specify a particular button. These tags can be any value in any order, giving you great flexibility. A common technique is simply to equate each tag to its cell's zero-based ordinal position within the matrix, either horizontally or vertically. Just click the Tags = Positions button to accomplish this for all the cells at once (in the current version of Interface Builder this is done for you automatically). Alternatively, you can select each button in turn and manually set the tags using the NSButton Info palette.

It is generally preferable to use tag or index values to identify user interface items in your code, instead of using their titles, because localization may change the titles.

This Step is remarkably simple. The interesting techniques and code appear in instructions 1.b., 2.b., d., f., and h., and 3.

1. Use Interface Builder to create a cluster of three radio buttons with a title. When you are done with this instruction 1, the group should look like *Screenshot 2-4*.

    a. From the Views palette, drag the two-button radio button group to the upper right area of the Buttons pane. Be careful to drop it on the pane, not the underlying window.

    b. Select the cluster by clicking anywhere in it. There are two ways in which you can create a third button. Using the NSMatrix Info palette, change the number of rows from 2 to 3 by typing in the Row/Col form at the bottom right. More easily, you can option-drag the bottom border of the matrix and watch the new buttons appear before your eyes. Using either technique, create a third button.

    c. Rename the three buttons "Democratic," "Republican," and "Socialist," respectively. (Bernie Sanders is the socialist Congressman from Vermont. Local tub thumping will hopefully be forgiven in these Vermont Recipes.)

    d. Drag the Message Text textbox onto the Buttons pane, placing it above and a little to the left of the radio buttons. The Aqua guides should be used to position the baseline of the text at the same level as the baseline of the Checkbox control's label. Then drag the radio button cluster in order to use the Aqua guides to position it the proper distance below the Message Text.

    e. Rename the textbox "Party Affiliation:" (note the trailing colon). You may have to choose Layout > Size to Fit to make the new text fully visible.

    f. In the NSTextField Info palette, uncheck the Selectable Option and make sure the Editable Option is also unchecked. This text field holds static text, that is, a heading that the user should not be able to edit, select or copy to the clipboard.

2. Use Project Builder to write the code.

    a. **User control outlet variable and accessors**. In the header file `MyWindowController.h`, declare a new outlet to access the radio button cluster, after the Music switch button group, as follows:

```
// Party radio button cluster
IBOutlet NSMatrix *partyRadioCluster;
```

Still in the header file, also declare the accessor for the outlet, after the Music switch button group, as follows:

```
// Party radio button cluster
- (NSMatrix *)partyRadioCluster;
```

In the source file `MyWindowController.m`, define the accessor after the Music switch button group, as follows:

```
// Party radio button cluster

- (NSMatrix *)partyRadioCluster {
    return partyRadioCluster;
}
```

b. **Data variable and accessors**. The variable that will hold the data value associated with the radio button cluster can be a simple C integer type, and its accessors and other methods can accept and return integers. However, the use of the C enumeration type to substitute constants for integers often promotes more understandable code.

In the header file `MySettings.h`, declare a new type immediately following the `#import` directive, as follows:

```
typedef enum {
    VRDemocratic,
    VRRepublican,
    VRSocialist
} VRParty;
```

Now you can declare the data variable as type `VRParty` instead of type `int`. You will see an example of how to do this in instruction 2.h., below.

When you use typedefs like this, you should generally prefix them with unique initials to avoid possible naming conflicts with third-party frameworks that you might use, just as you did earlier (and will continue to do) with the notification names. You should even follow this practice when naming your classes, although in Vermont Recipes the use of "`My`" as a class prefix (as in "MyDocument") is probably sufficient, if somewhat trite.

In the header file `MySettings.h`, declare the variable to access the data value associated with the radio button cluster, after the Music section, as follows:

```
// Party
VRParty partyValue;
```

In `MySettings.h`, also declare the corresponding accessor methods after the `Music` section, as follows:

```
// Party

- (void)setPartyValue:(VRParty)value;
- (VRParty)partyValue;
```

Turn to the source file `MySettings.m` and define these accessor methods after the `Music` section, as follows:

```
// Party

- (void)setPartyValue:(VRParty)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setPartyValue:partyValue];
    partyValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRPartyValueChangedNotification
object:self];
}

- (VRParty)partyValue {
    return partyValue;
}
```

c. **Notification variable**. Return to the header file `MySettings.h`, at the bottom of the file, to declare the notification variable used in the `set`... method, as follows:

```
// Party
extern NSString *VRPartyValueChangedNotification;
```

Turn back to the source file `MySettings.m`, near the top of the file, to define the notification variable, as follows:

```
// Party
NSString *VRPartyValueChangedNotification =
@"PartyValue Changed Notification";
```

d. **GUI update method**. Now you need a method to update the graphical user interface in response to this notification. First, however, since you are dealing with a new kind of control, you must devise a new generic method to update radio clusters. In the header file `MyWindowController.h`, declare the following method at the end of the Generic view updaters section:

```
- (void)updateRadioCluster:(NSMatrix *)control
setting:(int)value;
```

Because this is a generic method that must be able to update many different radio button clusters, the second parameter must be typed as an integer rather than the enumeration type you have created for this particular button.

Now define this method in the corresponding location in `MyWindowController.m`, as shown below. Here you see how matrix-based groups of controls can express their values, namely, by reporting the value of the tag of the currently-selected cell. By the same token, such a control's view is updated by selecting the cell whose tag corresponds to the integer value passed to it.

```
- (void)updateRadioCluster:(NSMatrix *)control
setting:(int)value {
    if (value != [[control selectedCell] tag]) {
        [control selectCellWithTag:value];
    }
}
```

Now you can declare the specific udpater in the header file `MyWindowController.h`, after the Music subsection of the Specific view updaters section, as follows:

```
// Party
- (void)updatePartyRadioCluster:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define this specific update method, after the Music subsection of the Specific view updaters section, as follows:

```
// Party

- (void)updatePartyRadioCluster:(NSNotification
*)notification {
    [self updateRadioCluster:[self
partyRadioCluster] setting:[[self mySettings]
partyValue]];
}
```

e. **Notification observer**. Now register the window controller as an observer of the notification that will trigger this updater, by inserting the following statement in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the Music registrations:

```
// Party
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updatePartyRadioCluster:)
name:VRPartyValueChangedNotification object:[self
mySettings]];
```

f. **Action method**. Next, you must add an action method. In the header file `MyWindowController.h`, after the Music subsection at the end of the file, add the following:

```
// Party
- (IBAction)partyAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action method, after the Music subsection at the end, as shown below. Here, again, you see the use of a cell's tag in a matrix-based user control. Since there may be multiple values associated with multiple radio buttons in the cluster, you must use a C `switch` statement or chained `if...else` statements.

```
// Party

- (IBAction)partyAction:(id)sender {
    [[self mySettings] setPartyValue:[[sender
selectedCell] tag]];
    switch ([[sender selectedCell] tag]) {
    case 0:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select Democratic
Party", @"Name of undo/redo menu item after
Democratic radio button was selected")];
        break;
    case 1:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select Republican
Party", @"Name of undo/redo menu item after
Republican radio button was selected")];
        break;
    case 2:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select Socialist
Party", @"Name of undo/redo menu item after
Socialist radio button was selected")];
        break;
    }
}
```

g. **Localizable.strings**. Update the `Localizable.strings` file with these undo and redo menu titles.

h. **Initialization**. To give you an example of how to use the enumeration type you defined in instruction 2.b., above, you will now set the value of the `partyValue` variable to VR`Republican`, which is one of the constants you defined there.

In the source file `MySettings.m`, add this line to the `initWithDocument:` method, after the initialization of `trianglePegsValue`:

```
[self setPartyValue:VRRepublican];
```

Also, return to Interface Builder to make sure the initial appearance of the radio button cluster is correct. First, select the Republican radio button. To do this, you must first select the entire cluster by clicking it, then double-click the Republican radio button. Then, in the Options area of the NSButtonCell Info palette's Attributes pane, check the Selected

checkbox. You will see the Republican radio button become selected immediately. It is not necessary to select the Democratic radio button and uncheck the Selected checkbox, because selecting the Republican radio button did this for you automatically. Now, when the application launches and a document opens, the Democratic radio button will appear selected by default.

i. **Data storage**. Deal now with persistent storage of the new data variables. In the source file `MySettings.m`, define this key at the end of the Keys and values for dictionary subsection:

```
// Party
static NSString *partyValueKey = @"PartyValue";
```

Add these lines at the end of the `convertToDictionary:` method:

```
// Party
[dictionary setObject:[NSString
stringWithFormat:@"%d", [self partyValue]]
forKey:partyValueKey];
```

Add these lines near the end of the `restoreFromDictionary:` method, before the call to `[[self undoManager] enableUndoRegistration]`:

```
// Party
[self setPartyValue:(VRParty)[[dictionary
objectForKey:partyValueKey] intValue]];
```

You will notice that you are saving and retrieving integer values here. You could instead add methods to the VRStringUtilities category that you implemented in *Step 3* in order to save the enumeration constants as strings, to make the file more readable for anyone examining it with a general file utility. This is left as an exercise for the reader; however, you can examine the project source files to see how we did it. Hint: In addition to writing the new methods in the VRStringUtilities category, you will have to call them in the `convertToDictionary:` and `restoreFromDictionary:` methods in `MySettings.m`.

j. **GUI update method invocation**. Finally, add an invocation of the control update method to the window controller. In `MyWindowController.m`, add the following call at the end of the `updateWindow` method :

```
// Party
[self updatePartyRadioCluster:nil];
```

3. You must inform the nib file of the new outlet and action you have created, then connect them to the new radio button cluster.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two header files in which you have created an outlet and an action, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

    b. Select the Instances tab and Control-drag from the File's Owner icon to the new radio cluster and click its outlet name, then click the Connect button in the Connections pane of the File's Owner Info palette. Take care that the entire radio cluster is enclosed in the square at the end of the line while you are drawing the connection. If you aren't careful, you will end up selecting one of the individual radio buttons, which is not what you want here.

    c. Control drag from the new radio cluster to the File's Owner icon and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSButton Info palette, then click the Connect button. Again, make sure you start the drag with the entire radio cluster, not one of the individual radio buttons.

4. Compile and run the application to test the interactions among the new radio buttons, explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly. If you haven't done so already, try changing a few of the controls you created in Steps 2 and 3 in between changing radio button selections, then undo all your actions in turn to confirm that multiple undo unwinds your actions in the same order no matter which controls or groups of controls are involved. If you took up the challenge to modify the VRStringUtilities category, use PropertyListEditor on a saved file to be sure the party affiliation was saved as a string value rather than an integer.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 6: A pop-up menu button

- Highlights:
  - Using an index and an enumeration type to manage a pop-up menu button

A pop-up menu button (also known as a pop-up list) is very similar in function to a radio button cluster. It allows you to choose one among multiple listed options, but the user has to click it to see all of the options it offers. The current choice appears in the button as its title, and when the list is expanded by holding down the mouse button over it, the list extends above and below the button as far as needed to allow the current choice to remain positioned where it was, on the button. According to the Mac OS 8 Human Interface Guidelines, radio button clusters should max out at "approximately seven" options. If you offer more than about seven options, consider using a pop-up menu button instead. You can also use a pop-up menu button when there are fewer than seven options, as space or design considerations may dictate.

Like a radio button cluster, only one variable is needed to store the value of a pop-up menu button. The value of the button can be specified in your code by a unique tag you assign, possibly using Interface Builder, as in the case of radio buttons, or as an integer index representing the zero-based positional index of the currently-selected menu item. It is customary, though not necessary, to use the C enumeration type in order to assign meaningful names to each of the integer values. There are times when tags are more useful, such as when you will continually re-alphabetize the menu items. Here, however, you will use index values to see how it is done.

Because the selected menu item in the pop-up menu button here will be identified by its index, it is not necessary to set its tag. Except for using the index instead of the tag, coding a pop-up menu button is virtually identical to coding a radio button cluster. As is the case with radio button clusters, it is possible to use the titles of the separate items to select and obtain the selection of a pop-up menu button, and Cocoa provides methods to convert among index, tag and title.

1. Use Interface Builder to create a pop-up menu button. When you are done with this instruction 1, it should look like *Screenshots 2-5a and b*.

    a. From the Views palette, drag the pop-up menu button to the Buttons pane, below the Party Affiliation radio button cluster. The pop-up menu button is distinguished by the double arrows at its right end. Be careful to drop it on the pane, not the underlying tab view.

    b. Select the button by double-clicking it. Three menu items appear superimposed over it. Add two more menu items to the list. You do this by switching to the Menus palette, then dragging the menu item named "Item" to the bottom of the list twice.

    c. Rename the five menu items "ME," "MA," "NH," "RI," and "VT," respectively. Select the menu item with a checkmark beside it and, in the NSMenuItem Info palette, select the Off radio button. Select the VT menu item last in order to leave it showing as the default selection when the list is collapsed, and select the On radio button in the Info palette. (Vermont is one of the five New England states. Tub thumping again.)

    d. Drag the Message Text textbox onto the Buttons pane, placing it to the left of the pop-up menu button.

    e. Rename the textbox "State:" (note the trailing colon), and choose Layout > Size to Fit to shrink the size of the item. Drag the textbox until it is aligned at the left edge below the title of the radio button cluster, using the Aqua guides for precise positioning. Then drag the pop-up menu so that its text baseline is on the baseline of the text field and it is the proper distance to the right of the text field.

    f. Using the Options area of the Info palette, make sure the text field is not Editable or Selectable, but Enabled.

    g. Select the pop-up button and choose Layout > Size to Fit to size the new button appropriately relative to its longest item, and position it in the pane according to the guidelines.

2. Use Project Builder to write the code.

    a. **User control outlet variable and accessors**. In the header file `MyWindowController.h`, declare a new outlet to access the pop-up menu button, after the Party radio button cluster section, as follows:

```
// State pop-up menu button
IBOutlet NSPopUpButton *statePopUpButton;
```

Still in the header file, also declare the accessor for the outlet, after the Party radio button cluster section, as follows:

```
// State pop-up menu button
- (NSPopUpButton *)statePopUpButton;
```

In the source file `MyWindowController.m`, define the accessor after the Party radio button cluster section, as follows:

```
// State pop-up menu button

- (NSPopUpButton *)statePopUpButton {
    return statePopUpButton;
}
```

b. **Data variable and accessors**. Create the variable that will hold the data value associated with the pop-up menu button and a related C enumeration type. In the header file `MySettings.h`, declare a new type following the party enumeration type, as follows:

```
typedef enum {
    VRMaine,
    VRMassachusetts,
    VRNewHampshire,
    VRRhodeIsland,
    VRVermont
} VRState;
```

Still in the header file, declare the variable to access the data value associated with the pop-up menu button, after the Party section, as follows:

```
// State
VRState stateValue;
```

In `MySettings.h`, also declare the corresponding accessor methods after the `Party` section, as follows:

```
// State

- (void)setStateValue:(VRState)value;
- (VRState)stateValue;
```

Turn to the source file `MySettings.m` and define these accessor methods after the `Party` section, as follows:

```
// State

- (void)setStateValue:(VRState)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setStateValue:stateValue];
    stateValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRStateValueChangedNotification
object:self];
}

- (VRState)stateValue {
    return stateValue;
}
```

c. **Notification variable**. In the header file `MySettings.h`, at the bottom of the file, declare the notification variable used in the `set`... method, as follows:

```
// State
extern NSString *VRStateValueChangedNotification;
```

In the source file `MySettings.m`, at the end of the notification definitions near the top, define the notification variable, as follows:

```
// State
NSString *VRStateValueChangedNotification =
@"StateValue Changed Notification";
```

d. **GUI update method**. You are dealing with a new kind of control, so you must again define a new generic method to update pop-up menu buttons. You will use the index of its selected menu item. In the header file `MyWindowController.h`, declare the following method at the end of the Generic view updaters section:

```
- (void)updatePopUpButton:(NSPopUpButton *)control
setting:(int)value;
```

Because this is a generic method that must be able to update many different pop-up menu buttons, the second parameter must be typed as an integer rather than any specific enumeration constant.

Now define this method in the corresponding location in `MyWindowController.m`, as shown below. Here you see that the button reports the value of the index of the currently-selected menu item. By the same token, such a control's view is updated by selecting the menu item whose index corresponds to the integer value passed to it.

```
- (void)updatePopUpButton:(NSPopUpButton *)control
setting:(int)value {
    if (value != [control indexOfSelectedItem]) {
        [control selectItemAtIndex:value];
    }
}
```

Now you can declare the specific udpater, after the Party subsection in the Specific view updaters section of `MyWindowController.h`, as follows:

```
// State
- (void)updateStatePopUpButton:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define this specific update method, after the Music subsection of the Specific view updaters section, as follows:

```
// State

- (void)updateStatePopUpButton:(NSNotification
*)notification {
    [self updatePopUpButton:[self statePopUpButton]
setting:[[self mySettings] stateValue]];
}
```

e. **Notification observer**. Register the window controller as an observer of the notification that will trigger this updater, by inserting the following statement in the

`registerNotificationObservers` method of the source file `MyWindowController.m`, after the Party registration:

```
// State
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateStatePopUpButton:)
name:VRStateValueChangedNotification object:[self
mySettings]];
```

f. **Action method**. Add an action method. In the header file `MyWindowController.h`, after the Party section at the end, add the following:

```
// State
- (IBAction)stateAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action method, after the Music section at the end of the file, as shown below. Here, again, you see the use of the menu item's index.

```
// State

- (IBAction)stateAction:(id)sender {
    [[self mySettings] setStateValue:[sender
indexOfSelectedItem]];
    switch ([sender indexOfSelectedItem]) {
    case 0:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select ME",
@"Name of undo/redo menu item after Maine pop-up
menu button was selected")];
        break;
    case 1:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select MA",
@"Name of undo/redo menu item after Massachusetts
pop-up menu button was selected")];
        break;
    case 2:
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Select NH",
@"Name of undo/redo menu item after New Hampshire
```

```
pop-up menu button was selected")];
        break;
    case 3:
        [[self document] undoManager]
setActionName:NSLocalizedString(@"Select RI",
@"Name of undo/redo menu item after Rhode Island
pop-up menu button was selected")];
        break;
    case 4:
        [[self document] undoManager]
setActionName:NSLocalizedString(@"Select VT",
@"Name of undo/redo menu item after Vermont pop-up
menu button was selected")];
        break;
    }
}
```

g. **Localizable.strings**. Update the `Localizable.strings` file with these undo and redo menu titles.

h. **Initialization**. Set the value of the `stateValue` variable to VR`Vermont`, which is one of the constants you defined in the `VRState` enumeration type. In the source file `MySettings.m`, add this line to the `initWithDocument:` method, after the initialization of `partyValue`:

```
[self setStateValue:VRVermont];
```

You already set up the Interface Builder selection in instruction 1.c., above.

i. **Data storage**. Deal now with persistent storage of the new data variables. In the source file `MySettings.m`, define this key at the end of the Keys and values for dictionary section:

```
// State
static NSString *stateValueKey = @"StateValue";
```

Add these lines at the end of the `convertToDictionary:` method:

```
// State
[dictionary setObject:[NSString
stringWithFormat:@"%d", [self stateValue]]
forKey:stateValueKey];
```

Add these lines near the end of the `restoreFromDictionary:` method, before the call to `[[self undoManager] enableUndoRegistration]`:

```
// State
[self setStateValue:(VRState)[[dictionary
objectForKey:stateValueKey] intValue]];
```

You will notice once again that you are saving and retrieving integer values here. You could instead add still more methods to the VRStringUtilities category that you implemented in *Step 3* in order to save the enumeration constants as strings, to make the file more readable for anyone examining it with a file reading utility. This is left as an exercise for the reader; however, you can examine the project source files to see how we did it.

j. **GUI update method invocation**. Finally, add invocations of the control update method to the window controller. In `MyWindowController.m`, add the following call at the end of the `updateWindow` method:

```
// State
[self updateStatePopUpButton:nil];
```

3. You must inform the nib file of the new outlet and action you have created, then connect them to the new pop-up menu button.

   a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two Vermont Recipes 2 header files in which you have created an outlet and an action, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

   b. Select the Instances tab and Control-drag from the File's Owner icon to the new pop-up menu button and click its outlet name, then click the Connect button in the Connections pane of the File's Owner Info palette.

   c. Control drag from the new pop-up menu button to the File's Owner icon and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSButton Info palette, then click the Connect button.

4. Compile and run the application to test the new pop-up menu button, and explore how undo and redo work and make sure changes can be saved to disk, restored, and reverted properly.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 7: A pull-down menu button

- Highlights:
  - Issuing commands from menu items in a pull-down menu button

A pull-down menu button (also known as a pull-down list) is an NSPopUpButton, but with a slightly different appearance, behavior and function. It has a single down-pointing arrow at the right end, signifying that it only pulls down, instead of two arrows pointing in opposite directions. Like a pop-up menu button, it allows you to choose one among multiple listed options. However, the button's title never changes, and the current choice thus doesn't appear in the button except when the list is expanded by holding down the mouse button over it. This behavior makes it suitable for choosing options in a constrained environment, where the context is fixed and readily apparent, and a constricted space, where a label can't appear beside it. It can also be used to choose and execute commands, since each menu item can have its own action method.

Here, you will create a pull-down menu button whose menu items simply beep the indicated number of times. The beeps are for demonstration purposes only, of course; they are stand-ins for whatever actions you might want to perform in your application. It is good to know, however, that there is an `NSBeep()` function in the AppKit, which can be used at any time to play an alert sound based on the user's system preferences. Since the pull-down menu is purely action-oriented, you will not need to provide for a data value, nor, therefore, undo, redo, save, read, or revert. All you need are outlets and actions.

This Step differs from the previous Step in that you will work with the two individual menu items, rather than the pull-down menu button itself.

1. Use Interface Builder to create a pull-down menu button. When you are done with this instruction 1, it should look like *Screenshot 2-6a and b*.

    a. From the Views palette, drag the pop-up menu button to the Buttons pane, below the State pop-up menu button. You will change the new button to a pull-down menu button in a moment.

    b. Select the button by double-clicking it. Three menu items appear superimposed over it. Select the topmost item for editing, and name it "Beeper." Name the next two items "Beep Once" and "Beep Twice" and set their States to Off. The topmost button will become the fixed title when you change the button to a pull-down button. (Should you want to retitle it later, you will have to turn it back into a pop-up menu button temporarily.)

    c. Click outside the button to dismiss the menu items, then select the button itself again. In the NSPopUpButton Info palette, click the PullDown radio button. You will see the button change appearance to that of a pull-down menu button.

    d. If necessary, choose Layout > Size to Fit to size the new button appropriately relative to its longest item, and position it in the pane according to the Aqua guides.

2. Use Project Builder to write the code.

    a. **User control outlet variable and accessors**. If you were planning to implement some means to disable the two menu items, Beep Once and Beep Twice, you would need to create two new outlets in MyWindowController, using the same technique you used in *Step 5*, instruction 2.a.. You might call these new outlets `beep1MenuItem` and `beep2MenuItem`. Their type would be NSMenuItem. You would need to declare the variable and the accessor method in the header file, and define the accessor method in the source file.

    However, since you aren't going to let the user disable these menu items, you don't need outlets or accessor methods for them. You don't need to create a data variable, because there is no data value associated with the pull-down menu button, nor do you need a C enumeration type. For the same reason, you don't need notifications to update the control or an observer. All you need are action methods.

    b. **Action method**. You will create two action methods, one to carry out the commands issued by each of the two menu items in the pull-down menu button. In the header file `MyWindowController.h`, after the State subsection at the end, add the following:

```
// Beeper
- (IBAction)beep1Action:(id)sender;
- (IBAction)beep2Action:(id)sender;
```

In the source file `MyWindowController.m`, define the action methods, after the State section at the end, as shown below. `NSBeep()` is a Cocoa function that beeps.

```
// Beeper

- (IBAction)beep1Action:(id)sender {
    NSBeep();
}

- (IBAction)beep2Action:(id)sender {
    NSBeep();
    NSBeep();
}
```

There are no undo or redo strings to be updated in the `Localizable.strings` file, no data to initialize or store, and no control update methods to invoke.

3. You must inform the nib file of the new actions you have created, then connect them to the new pull-down menu button's menu items.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the header file in which you have created the actions, `MyWindowController.h`, then click the Parse button.

    b. Double-click the pull-down menu to reveal the two menu items.

    c. Control drag from each of the new menu items to the File's Owner icon in turn, and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSButton Info palette, then click the Connect button.

4. Compile and run the application to test the new pull-down menu button, listening for the beeps.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

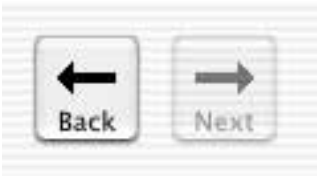
By Bill Cheeseman

June 20, 2001 - 9:00 AM

Recipe 2: User controls—Buttons

Step 8: Bevel buttons to navigate a tab view

- Highlights:
  - Placing an image and text on a bevel button
  - Creating a navigation button that appears on every tab view item in a tab view
  - Using a delegate method to disable navigation buttons when the first or last tab view item is selected



Bevel buttons are ordinary buttons, except that they are square and come in any size. In Mac OS X, the bevel height is always the same, no matter the size of the button. Bevel buttons usually hold an image or icon, and they can also contain text, usually placed below the image. The Aqua Human Interface Guidelines don't specify a font size for the label but they do note that the Finder uses a 10-point font.

In this Step, you will create two bevel buttons that behave as push buttons. They highlight momentarily when clicked but do not alter their appearance, as a "sticky" button does. They will function as navigation buttons to take the user back to the previous pane (tab view item) or forward to the next pane.

There are two interesting techniques used in this Step.

First, a single pair of navigation buttons will be made to appear as if they were different buttons in each of the tab view items, by the simple expedient of dropping them into the window instead of onto the tab view. They could be placed to one side of the tab view, but it can be made to appear as if there are buttons in every tab view item by placing them in the same space occupied by the tab view and layering them in front of the tab view.

Second, a delegate method is implemented, taking advantage of the ability of NSTabView to tell its delegate when a new tab view item has been selected and is about to come to the front. You need some means to disable the Back button when the first tab view item is selected, and to enable it when another tab view item is selected. Similarly, you need a means to disable the Next button when the last tab view item is selected, and to enable it when another tab view item is selected. Because tab view items can be selected in any of several ways—for example, by clicking any tab or by clicking either navigation button—you might think that disabling and enabling the navigation buttons would require adding code in several places. However, the power of delegation can be harnessed to do it all in a single, short method in the window controller. This will be your first significant encounter with a delegation method provided for in Cocoa. Pay close attention, because it is an extraordinarily useful technique, used heavily throughout Cocoa.

1. Use Interface Builder to create two bevel buttons. When you are done with this instruction 1, they should look like *Screenshot 2-7* (but their images may be different, depending on where you find suitable arrow images).

    a. From the Views palette, drag the square-shaped, round-cornered bevel button bearing a Mac face to the document window twice. Drop each of them in the area occupied by the tab view, near its bottom right corner. This time, contrary to your practice in all of the previous Steps, you should be careful to drop them into the window, not onto the currently selected tab view item. This is easy; just click once in an empty area of the window outside of the tab view, to make sure that neither the tab view nor either of its tab view items is selected, then perform the drags. To make sure the buttons do not become hidden behind the tab view, choose Layout > Bring to Front immediately, while each button is selected, before you click on something else. (If you do accidentally click on something else and the new buttons disappear behind the tab view, you can get them back by selecting the tab view and choosing Layout > Send to Back to layer it behind the new buttons.) Place the buttons side by side in the lower right corner of the pane. Their Mac faces will have disappeared along the way. Because they reside in the window, not the tab view, you will not see Aqua guides to help you position them relative to the lower right edges of the tab view. However, if you hold down the Option key while moving them, you will see their distances from those edges and can nudge them into position using the arrow keys (20 pixels from each edge). The Aqua guides will help you determine how far apart they should be.

    b. Find or create two images of left and right arrows, respectively. They can be TIFF or PNG images (or GIF, which have more limited capability and present potential licensing issues), or any of a number of other graphic types. Consult the Aqua Human Interface Guidelines regarding the dimensions of the images. If you have nothing more suitable handy, use the two TIFF images in the project files that come with *Vermont Recipes*. To install the images into your project, first save or drag the image files into the Vermont Recipes 2 project folder. Then choose Project > Add Files… in Project Builder and add them to the Resources group in the Groups & Files pane of the main project window.

If the images might require localization, they can be placed in the appropriate language subfolder in the project folder, instead, and a localization contractor can substitute other images in other language subfolders.

(In Mac OS X Public Beta and earlier, it was feasible to store images in the nib file. It is now recommended that they always be stored in the project folder.)

c. By adding the image files to the project, you made them automatically available to Interface Builder in the Images tab of the `MyDocument.nib` window. Drag the left arrow and right arrow images in turn from the Images pane of the `MyDocument.nib` window and drop them onto the left or right bevel button, respectively. They will center themselves.

d. In the Attributes pane of the NSButton Info palette, first make sure the lower right button is selected in the Icon Position area of the Info palette in order to move the image upwards to make room for the text you are about to add, and that the centered button is selected in the Alignment area. Then type "Back" as the title for the left button and "Next" as the title for the right button. The text will appear in the buttons.

e. Select the "Back" and "Next" text for editing in each button in turn. Choose Format > Font > Show Fonts to open the Fonts dialog, and verify that the font size of the button text is 10 points.

f. If either of the buttons has changed size unexpectedly, select it, then select the Size pane of the NSButton Info palette. Drag the resizing handles until each button is 40 by 40 pixels, or larger if your left and right arrows are large. Leave at least 6 pixels around each image within its button, and make sure the buttons are the same size

2. Use Project Builder to write the code.

a. **User control outlet variable and accessors**. Create two new outlets in MyWindowController using the same technique you used in *Step 5*, instruction 2.a., but call these new outlets `backButton` and `nextButton`. Their type should be `NSButton`. You will need to declare the variable and the accessor method for each in the header file, and define the accessor methods in the source file.

You don't need to create a data variable in MySettings, because there is no data value associated with these buttons. For the same reason, you don't need a C enumeration type, notifications to update the buttons, or observers.

b. **Action method**. You want to create two action methods, one to carry out the commands issued by each of the two navigation buttons. The action methods will tell the tab view to select the previous or next pane, respectively.

But you never created an instance variable to enable you to talk to the tab view. You'll have to do that now, along with a related accessor method.

In the header file `MyWindowController.h`, above the `myCheckbox` variable declaration, add the following:

```
IBOutlet NSTabView *myTabView;
```

Also in the header file, create the accessor method above the `myCheckbox` accessor declaration at the top of the Accessor methods and conveniences section, as follows:

```
- (NSTabView *)myTabView;
```

In the source file `MyWindowController.m`., above the `myCheckbox` accessor declaration at the top of the Accessor methods and conveniences section, add this definition:

```
- (NSTabView *)myTabView {
    return myTabView;
}
```

c. Now you can create the navigation button action methods. In the header file `MyWindowController.h`, after the Beeper section at the end, add the following:

```
// Navigation
- (IBAction)backAction:(id)sender;
- (IBAction)nextAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action methods, after the Beeper section at the end, as shown below.

```
// Navigation

- (IBAction)backAction:(id)sender {
    [[self myTabView]
selectPreviousTabViewItem:sender];
}

- (IBAction)nextAction:(id)sender {
    [[self myTabView]
selectNextTabViewItem:sender];
}
```

NSTabView's `selectPreviousTabViewItem` and `selectNextTabViewItem` methods do nothing if there is no previous or next tab item, so you never have to worry about an error if the Back button is clicked while the first tab item is selected, nor if the Next button is clicked while the last tab item is selected.

Note that NSTabView implements a full set of methods for navigation among panes in a tab view, including methods to select the first tab item, the last tab item, and any tab item by its index.

d. Although the action methods will not cause an error if an attempt is made to navigate past the first or last tab view item, it is good user interface design to disable either button when the user can't navigate any farther in the one direction or the other. To do this, you will implement in MyWindowController a delegate method provided for in NSTabView, `willSelectTabViewItem:`. If you haven't worked with the delegation capability of Objective-C before, you will find it to be an eye-opening and powerful feature of the language.

   This delegate method is anticipated in NSTabView. Whenever the user selects a new tab view item by any means—clicking a tab, choosing a pull-down menu item, or clicking a button, for example—a method built into NSTabView is invoked that selects the chosen tab view item. This method, among other things, checks to see whether a delegate has been appointed for NSTabView and, if so, whether that delegate implements the `willSelectTabViewItem:` delegate method. If neither condition is met, the delegate method is not called. If, however, NSTabView discovers that a delegate has been appointed and that the delegate implements this method, then it is invoked. You, the developer, can use Interface Builder to appoint a delegate for NSTabView (here, you will appoint MyWindowController as the delegate), and you can implement the delegate method in the delegate and code it to do anything that will be useful when a user chooses a new tab view item.

   What your delegate method will do here, of course, is to test whether the newly-selected tab view item is the first or last and disable or enable the two navigation buttons accordingly. Because of the power of delegation, you will only have to write this one simple method, and Cocoa will see to it that it is invoked every time the user chooses a new tab view item by any means.

   In the source file `MyWindowController.m`, define the delegate method at the end of the Action methods section, as follows:

```
- (void)tabView:(NSTabView *)theTabView
willSelectTabViewItem:(NSTabViewItem
*)theTabViewItem {
    if (theTabView == [self myTabView]) {
        [[self backButton] setEnabled:([theTabView
indexOfTabViewItem:theTabViewItem] > 0)];
        [[self nextButton] setEnabled:([theTabView
indexOfTabViewItem:theTabViewItem] + 1 <
[theTabView numberOfTabViewItems])];
    }
}
```

You do not have to declare a delegate method in the header file, because Cocoa declares and invokes it for you.

That's all there is to it, except for connecting things up in Interface Builder. There are no undo or redo strings to be updated in the `Localizable.strings` file, no data to initialize or store, and no control update methods to invoke.

3. You must inform the nib file of the new outlets and actions you have created, then connect them to the associated objects.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the header file in which you have created outlets and actions, `MyWindowController.h`, then click the Parse button.

    b. Control-drag from the tab view to the File's Owner icon in the `MyDocument.nib` window, select the built-in delegate outlet in the NSTabView Info palette, and click the Connect button. When dragging from the tab view, take care not to drag from the current tab view item, but from the tab view. It is best to start the drag from the open area beside the first or last tab above the top of the tab view.

    c. Control-drag from the File's Owner icon to the tab view, click its outlet name in the NSButton Info palette, `myTabView`, then click the Connect button. When dragging to the tab view, be careful to end on the tab view , not the current tab item view. The easiest way to do it correctly is to end the drag in the open area beside the first or last tab above the top of the tab view.

    d. Control-drag from the File's Owner icon to each of the new bevel buttons in turn, and click its outlet name in the Info palette, `backButton` or `nextButton`, then click the Connect button.

    e. Control drag from each of the new bevel buttons to the File's Owner icon in turn, and click

the target in the left pane and the appropriate action, `backAction` or `nextAction`, in the Info palette, then click the Connect button.

f. If the Buttons tab is going to be the initially-selected tab when the document window is opened, the Next button should be disabled at the outset as long as the Buttons pane is the last tab view item. Select the Next button and, in the NSButton Info palette, uncheck Enabled. You will come back to change this later, when you add another tab view item.

g. Finally, since you have now fully populated the Buttons pane of the document window, step back and consider its overall arrangement. It's pretty ugly, but that's mainly because this is a set of disparate examples; there isn't a unifying theme or function to the pane. However, there is one thing you should do to finish your effort to comply with human interface guidelines. Mac OS X dialogs are supposed to gravitate towards the center, horizontally. So drag the Checkbox switch and the Pegs group box to the right until their right edges align with the right border of the Music group box; the Aqua guides make this especially easy. It's still ugly, but you'll do better in your real applications.

4. Compile and run the application. To test the new Back button, click it and note that the Text Boxes pane is selected. Then click the Next button on that pane and note that the Buttons pane is selected again. In each case, confirm that the navigation buttons are disabled and enabled appropriately.

You are now done with *Recipe 2*. There are still more variants of buttons you can use in your application, but this recipe has given you enough of a head start that you will be able to figure out how to create additonal buttons in your application.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 3: User controls—Sliders

Download the project files for Recipe 3 as a disk image and install them

Download a pdf version of Recipe 3

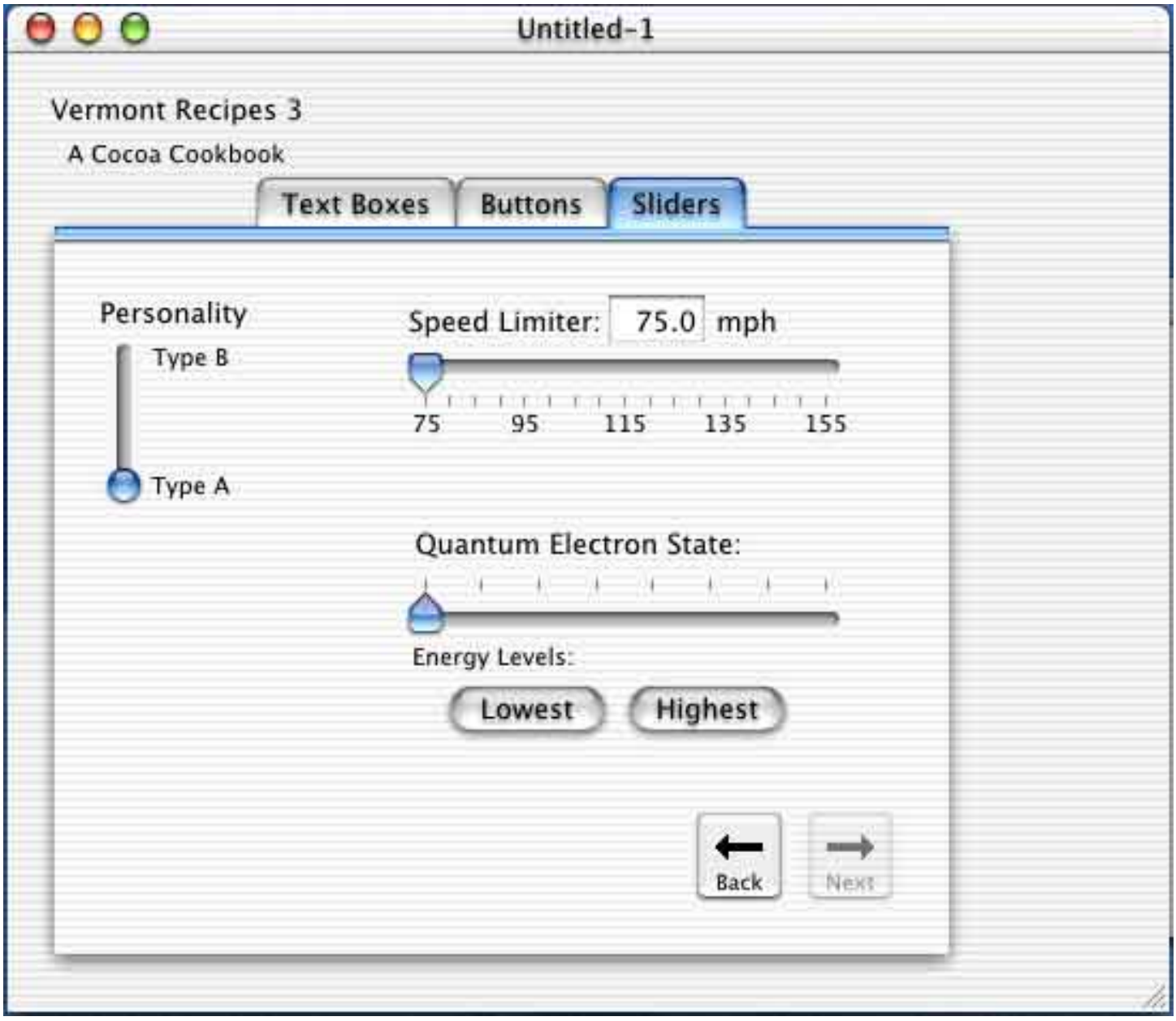
This Recipe is the second in a series of Recipes dealing with user controls. In *Recipe 2*, you implemented a bunch of buttons. Here, you will learn about some slick sliders and get a brief introduction to text fields and sheets.

Sliders are fun, in addition to being useful. They have a much more interactive and realistic feel than other controls, probably because of their graphical complexity and the continuous visual feedback they give you as you drag them back and forth or up and down. Also, they give you an instantaneous sense of scale that pure numbers lack. If a slider is set towards one end, you immediately sense that it represents something that is at, say, a fifth or a quarter of the range of available values. Where absolute precision doesn't matter, it is much easier to set a slider approximately where you want it than it is to type in a number. With the addition of tick marks and the use of a discrete slider—one that snaps to integer values, say—accuracy is also easily achieved.

Sliders provide the best of both worlds if they are linked to a text box. The text box can report the slider's setting in exact numbers, and you can type a specific number into the text box and see the slider instantly snap to that setting. Cocoa provides a linking mechanism that causes the numbers in the text box to spin by very fast as you drag the slider, creating an extraordinarily responsive feel. Sliders can also be linked to buttons and other user controls; for example, you can provide buttons that instantly set the slider to its minimum and maximum positions or, say, to one-third and two-thirds positions. In this Recipe, you will create all of these examples.

Before turning to Step 1, you should prepare your project files for this *Recipe 3*. You do not have to follow all of the procedures you went through to prepare for the previous Recipe. You can leave all old references to Vermont Recipes alone, for example, because that will be the name of the application in all subsequent Recipes. You should, however, update the application and document signatures to **`VRa3`** and **`VRd3`**, respectively, and change the version number to **`1.0.0d3`**, in the Targets pane and to change the version number to **1.0.0d3** in the InfoPlist.strings file, as detailed in *Recipe 2, Step 1*. It would also be a good idea to change the `currentMyDocumentVersion` variable in `MyDocument.m` to 3.

Screenshot 3-1: The Vermont Recipes 3 application



Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 3: User controls—Sliders

Step 1: A simple slider

- Highlights:
  - Adding a tab view item using Interface Builder
  - Presenting a simple document-modal sheet
  - Using the NSString `localizedStringWithFormat:` class method to concatenate strings and to format numbers using localized formatting conventions

In *Step 1*, you will create a simple vertical slider that returns a floating point value within a defined range after you drag it to a new setting. It will be adorned with a title and labels at top and bottom, but it will have no tick marks. The slider will describe a personality in a continuous range from Type A to Type B (never mind whether psychologists recognize a B+ or an A- personality type). You will add this control using the roadmap that was adopted in *Recipe 2, Step 3*.

The slider will be placed in a new tab view item that will hold the various sliders you will create in this *Recipe 3*. The technique for adding a tab view item in Interface Builder is described in instruction 1.a., below. One interesting thing to note is that you do not have to update the code for disabling and enabling the navigation buttons you created in *Recipe 2, Step 8*—they work as is, without revision, no matter how many tab view items you may add.

In order to let you verify that the slider returns an appropriate value when you are done dragging it, the application will present a sheet reporting its final value. This is not something you would necessarily want to do in a real-world application; instead, you might provide a text field to present the value, as you will do later in *Step 2*, or you might simply rely on the slider itself as a sufficient presentation of the data. Here, however, a sheet will give you an introduction to presenting alerts in Cocoa, in addition to giving you some comfort that the slider is working correctly. The code for this appears in instruction 3., below. It makes use of an

important Cocoa method, the NSString `localizedStringWithFormat:` method, which you will use frequently throughout your applications.

1. Use Interface Builder to add a tab view item to the existing tab view and create a new vertical slider with a title and labels. When you are done with this instruction 1, the group should look like that shown in *Screenshot 3-2*.

    a. Using `MyDocument.nib` in Interface Builder, select the tab view (not a tab view item) in the document window by clicking in the empty space on either side of the existing tabs. In the NSTabView Info palette, type 3 in the Number of Items box and hit the Enter key. A third tab appears to the right of the two existing tabs. Double-click twice on the new tab to to edit its label and type **`Sliders`**. When you hit the Enter key, the tab resizes so that the new label will fit, and the three tabs center themselves as a group. Click once outside the tab view, and the two navigation buttons you created in Recipe 2 appear. The Next button remains disabled; you might as well leave it that way for now.

    b. With the new Sliders tab view item selected, drag the rightmost vertical slider from the More Views palette onto the Sliders pane and place it in the upper left area of the pane. In the NSSlider Info palette, leave all but one of the Attributes set as you find them, including a range from 0.0 to 100.0 and no tick markers. However, you should uncheck the Continuous option. There is no need for this control to send its action method continuously as the slider is dragged up and down; it only needs to set its associated variable when the drag is completed. Also, in instruction 3., below, you will add a statement to the slider's action method to present a sheet, and you only want the alert to appear when the drag is completed.

    c. Drag the Messsage Text item from the Views palette and place it above the slider. Type **`Personality`** and choose Layout > Size to Fit. Make sure the Editable and Selectable checkboxes in the Options area of the NSTextField Info palette are unchecked and that the Enabled checkbox is checked.

    d. Drag the Informational Text item from the Views palette and place it to the right of the slider at its bottom, using the Aqua guides to position it properly. Rename it **`Type A`** and choose Layout > Size to Fit. For vertical sliders, the minimum value is always at the bottom. Uncheck the Editable and Selectable checkboxes and check the Enabled checkbox, if necessary.

    e. Duplicate the Type A label by Option-dragging it to the top of the slider, and rename it **`Type B`**.

    f. Reposition all the new items to comply with human interface guidelines. If you select the slider and its minimum and maximum labels, then drag them, Apple guides would normally appear when the group is centered the proper distance beneath the group's title, if the title were long enough. Here, the title is too short, and the slider and its labels want to snap to the left or right guides. To center them, choose Layout > Guides > Disable Aqua Guidelines, then use the arrow keys to nudge them until they are centered under the title.

    g. Select all the new items, then choose Layout > Group in Box, and use the NSBox Info palette

to hide the title and select the no border box type. Now the new items can be dragged as a group without selecting all of them individually.

2. Use Project Builder to write the code required to make the new slider work.

    a. **User control outlet variable and accessors**. In the header file `MyWindowController.h`, declare an outlet variable after the existing variable declarations, as follows:

```
// Personality slider
IBOutlet NSSlider *personalitySlider;
```

    Still in `MyWindowController.h`, also declare an accessor method for the outlet at the end of the Accessor methods and conveniences section, as follows:

```
// Personality slider
- (NSSlider *)personalitySlider
```

    Turn to the source file `MyWindowController.m` and define the accessor method at the end of the Accessor methods and conveniences section, as follows:

```
// Personality slider

- (NSSlider *)personalitySlider {
    return personalitySlider;
}
```

    b. **Data variable and accessors**. In the header file `MySettings.h`, declare a new variable after the existing variable declarations, as shown below. The data type represented by a slider is a C float type.

```
// Personality
float personalityValue;
```

    In `MySettings.h`, also declare the corresponding accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Personality

- (void) setPersonalityValue:(float)value;
- (float) personalityValue;
```

In the source file `MySettings.m`, define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Personality

- (void) setPersonalityValue:(float)value {
    [[[self undoManager] prepareWithInvocationTarget:self]
setPersonalityValue:personalityValue];
    personalityValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRPersonalityValueChangedNotification
object:self];
}

- (float)personalityValue {
    return personalityValue;
}
```

c. **Notification variable**. Return to the header file `MySettings.h`, at the bottom of the file, to declare the notification variable, as follows:

```
// Personality
extern NSString
*VRPersonalityValueChangedNotification;
```

Turn back to the source file `MySettings.m`, near the top of the file, to define the notification variable, as follows:

```
// Personality
NSString *VRPersonalityValueChangedNotification =
@"PersonalityValue Changed Notification";
```

d. **GUI update method**. In the header file `MyWindowController.h`, declare a user interface update method at the end of the Specific view updaters section, as follows:

```
// Personality
- (void)updatePersonalitySlider:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define this specific update method at the end of the Specific view updaters section, as follows:

```
// Personality

- (void)updatePersonalitySlider:(NSNotification
*)notification {
    [self updateSlider:[self personalitySlider]
setting:[[self mySettings] personalityValue]];
}
```

As you see, a new generic updater is required for sliders. In the header file `MyWindowController.h`, declare a user interface update method at the end of the Generic view updaters section, as follows:

```
- (void)updateSlider:(NSSlider *)control
setting:(float)value;
```

In the source file `MyWindowController.m`, define this update method at the end of the Generic view updaters section, as follows:

```
- (void)updateSlider:(NSSlider *)control
setting:(float)value {
    if (value != [control floatValue]) {
        [control setFloatValue:value];
    }
}
```

e. **Notification observer**. Register the window controller as an observer of the notification that will trigger the updater method, by inserting the following statement in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the existing registrations:

```
// Personality
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updatePersonalitySlider:)
name:VRPersonalityValueChangedNotification
object:[self mySettings]];
```

f. **Action method**. In the header file `MyWindowController.h`, at the end of the Action methods section, add the following:

```
// Personality
- (IBAction)personalityAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action method at the end of the Action methods section, as follows:

```
// Personality

- (IBAction)personalityAction:(id)sender {
    [[self mySettings] setPersonalityValue:[sender
floatValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Personality",
@"Name of undo/redo menu item after Personality
slider was set")];
}
```

g. **Localizable.strings**. Update the `Localizable.strings` file by adding "Set Personality."

h. **Initialization**. Leave the Personality data variable uninitialized. Objective-C will initialize it to `0`, which in this case stands for a Type A personality.

i. **Data storage**. In the source file `MySettings.m`, define the following key in the Keys and values for dictionary section:

```
// Personality
static NSString *personalityValueKey =
@"PersonalityValue";
```

Immediately after that, add these lines at the end of the `convertToDictionary:` method:

```
// Personality
[dictionary setObject:[NSString
stringWithFormat:@"%f", [self personalityValue]]
forKey:personalityValueKey];
```

And add these lines near the end of the `restoreFromDictionary:` method, before the call to `[[self undoManager] enableUndoRegistration]`:

```
// Personality
    [self setPersonalityValue:(float)[[dictionary
objectForKey:personalityValueKey] floatValue]];
```

j. **GUI update method invocation**. Finally, in `MyWindowController.m`, add the following call at the end of the `updateWindow` method :

```
// Personality
[self updatePersonalitySlider:nil];
```

3. To enable you to verify that the slider set the data variable to an appropriate value, you will now modify the slider's action method so that it presents a sheet reporting the value whenever the value is changed. This is why you unchecked the Continuous attribute of the slider in instruction 1.b., above: you don't want the alert to be presented repeatedly as you drag the slider, but only when the drag is completed.

   Go back to the slider's action method definition in `MyWindowController.m` and add the following statements calling the Cocoa `NSBeginAlertSheet()` function, at the end of the method. This function was introduced in Developer Preview 4 and now, in the final release of Mac OS X, replaces the deprecated `NSRunAlertPanelRelativeToWindow()` function. Documentation for the new function is sparse at this time. It is declared in NSPanel.h, where you will find some cryptic comments regarding its use, and there is some relevant discussion in the Developer Preview 4 and Public Beta AppKit Release Notes, under the heading "Document-Modal API." The invocation here is the simplest possible, providing information to the user but offering no alternative course of action. It makes no use of the modal delegate or other available parameters, but simply goes away when the user clicks OK. You will learn about these more complex features of sheets, including how to provide multiple buttons and respond to the user's selection, in *Recipe 4*. For now, just passing `NULL` or `nil` in most of the parameters does the trick.

```
alertMessage = [NSString
localizedStringWithFormat:NSLocalizedString(@"The
personality type you set is %f in a range from 0 to 100.",
@"Message text of alert posed by Personality slider to
report value set by user"), [[self mySettings]
personalityValue]];

alertInformation = NSLocalizedString(@"0 is Type A, 100 is
Type B.", @"Informative text of alert posed by Personality
slider to report value set by user");

NSBeginAlertSheet(alertMessage, NULL, NULL, NULL, [self
window], nil, NULL, NULL, NULL, alertInformation);
```

You must also declare the two local variables, `alertMessage` and `alertInformation`, at the beginning of the method, as follows:

```
NSString *alertMessage;
NSString *alertInformation;
```

The `NSBeginAlertSheet()` function poses the alert as a Mac OS X sheet, so the sheet remains attached to the window whose slider was set. If many windows are open at once, the user will have no doubt about which window's slider value the alert is reporting.

Note the use of the NSString `localizedStringWithFormat:` class method. This is a method you will make heavy use of in your Cocoa applications, as you will its companion method, `stringWithFormat:`. They provide concatenation of multiple string values and, in the case of the former, localization of many data types. Just pass in a formatting string followed by any number of values or variables, separated by commas, and include placeholders for each of the values in the formatting string (such as `%f` for a floating point value). The placeholders will be filled by string versions of the values in the order given. The `localizedStringWithFormat:` method is appropriate when one or more of the values you want to convert is a numeric value and you want to use the numeric formatting conventions of the particular computer's locale as set in System Preferences (in many countries, for example, the thousands separator is a period and the decimal separator is a comma).

You are already familiar with the `NSLocalizedString()` function from *Recipe 1, Step 3.1.4*. You have used it repeatedly in *Recipe 2* and *Recipe 3* when naming undo and redo strings using localization key-value pairs from the `Localizable.strings` file. Here, you use it for a similar purpose, providing internationalized strings for the message and information text in a sheet. Be sure to put the key-value pairs in the `Localizable.strings` file now, as shown below. Notice that the printf-style placeholder `%f` continues to serve its purpose even though it is passed through the `Localizable.strings` file (assuming the localization contractor includes it in the new localized string).

```
/* Alert strings */

/* Personality slider alert r3s1 */
/* Message text of alert posed by Personality slider to
report value set by user */
"The personality type you set is %f in a range from 0 to
100." = "The personality type you set is %f in a range
from 0 to 100.";
/* Informative text of alert posed by Personality slider
to report value set by user */
"0 is Type A, 100 is Type B." = "0 is Type A, 100 is Type
B.";
```

4. Inform the nib file of the new outlet and action you have created in the code files, then connect them to the new slider.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

    b. Select the Instances tab and Control-drag from the File's Owner icon to the new slider and click its outlet name, then click the Connect button in the Connections pane of the File's Owner Info palette.

    c. Control drag from the new slider to the File's Owner icon and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSSlider Info palette, then click the Connect button.

    d. In the Attributes pane of the NSSlider Info palette, set the current value of the slider to 0, since the corresponding data variable was left at the default initialization value of 0.

Compile and run the application to test the slider. Be sure to explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly.

Also, open several windows at once and exercise the slider in each, to confirm that you can leave the sheet open and still switch to another window and open its sheet, as well. The sheet is said to be "document modal," meaning that you can't do any work in its document until the sheet is dismissed, but you can do work in other documents without dismissing it.

If you harbor some lingering doubts about the internationalized key-value pairs used in the Personality slider alert, you do not have to travel to an exotic country or hire a localization contractor to test it. Just change the value associated with the key in your own `Localizable.strings` file by adding some garbage text, then run the application and set a new Personality value. You will now see your garbage text in the alert that is presented. (Don't forget to change the value in the `Localizable.strings` file back to its original text after completing this experiment.)

Finally, test the navigation buttons that you added in *Recipe 2, Step 7*. Confirm that they disable and enable themselves at the right times, even though you have added a tab view item in this step without updating the code for the navigation buttons.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

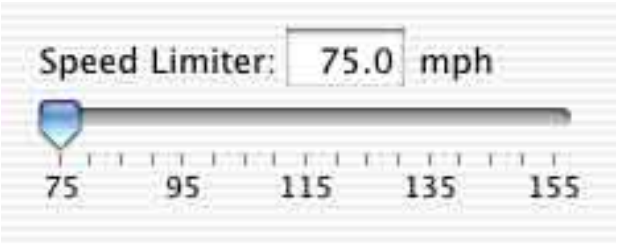
August 1, 2001 - 6:00 PM

Recipe 3: User controls—Sliders

Step 2: A slider with an interactive text field

- Highlights:
  - Setting an editable text field from a slider's value continuously as the slider is dragged
  - Setting a slider from an editable text field
  - Using the NSString `localizedStringWithFormat:` class method to format numbers using localized formatting conventions
  - Using an NSFormatter object to constrain text entry to a floating-point value within a predefined range

In this Step, you will create a horizontal slider with tick marks. It will return a floating point value within a predefined range, as in *Step 1*, but the range does not start at 0.0 and must therefore be initialized. The slider will allow you to set the automatic speed limiter on your Porsche Boxster to a safe and conservative value (Boxster available separately).



The slider in this Step is associated with a text field, which is a much more useful device than the sheet you used in *Step 1* to communicate the final value set by the slider. The value in the text field will always reflect the value of the slider, changing rapidly as the user drags the slider back and forth. For those who prefer to type, a value can be entered in the text field and the slider will snap to the indicated value when the Enter key is pressed.

This is your first editable text field, and it no doubt occurs to you that there may be a problem if the user types something into the field that isn't a number, such as the word "fast." The text field is supposed to

accept only floating point numbers within a predefined range. You will therefore associate a Cocoa NSFormatter object with the field to apply the desired constraints on text entry. A formatter can be very complex, but in this introduction to the topic you will find it very easy to use the formatters supplied with Cocoa.

The formatter will cause the text field to beep if the user attempts to enter an illegal value (an out-of-range value or text that isn't a valid number). This is adequate for experienced users, but you might not want to leave any of your users guessing. You should consider presenting an alert to explain what the problem is and provide an easy way out. However, the alert you want here is considerably more complex than the simple sheet you used in *Step 1*. It should provide additional buttons to let the user cancel an illegal entry or subsitute the minimum or maximum value for an out-of-range value. You will therefore defer creating such an alert until *Recipe 4*, concentrating for now on getting the formatter working.

1. Use Interface Builder to create a new horizontal slider with a title, tick marks and labels, along with an associated text field and label. When you are done with this instruction 1, the group should look like that shown in *Screenshot 3-3*.

    a. With the Sliders tab view item selected, drag the topmost horizontal slider from the More Views palette onto the Sliders pane and place it in the upper right area of the pane. In the NSSlider Info palette, type **`75.0`** as the Minimum Value and **`155.0`** as the Maximum Value (the current value should automatically assume a value within that range, but type **`75.0`** for the Current Value, if necessary); check the Continuous checkbox; set the Number of Markers to 17 with Position Below; and uncheck Marker Values Only. This slider must be a "continuous" slider, invoking its action method repeatedly as it is dragged back and forth, in order to update the associated text field while the drag is under way; you will see how this is done in instruction 3., below.

    b. Drag Message Text from the Views palette to the Sliders pane twice, naming one "Speed Limiter:" (note the colon) and the other "mph," and position them above the slider with space between them for a small text field. Choose Layout > Size to Fit, if necessary to make all of the longer text item visible and to shrink the shorter text item, and line them up on the same text baseline. Drag Informational Text to the Sliders pane once, type 75, choose Layout > Size to Fit, and position it beneath the leftmost tick mark. Create additional mile-per-hour labels by Option dragging to duplicate the "75" label, typing **`95`**, **`115`**, **`135`**, and **`155`** and choosing Layout > Size to Fit on each, and position each beneath the appopriate tick mark (each tick mark denotes a 5-mph increment). An easy way to get started with alignment is to use the Aqua guides to position the first and last label at the left and right ends of the slider, respectively, then select all of the sliders, choose Tools > Alignment to open the Alignment palette, and select the leftmost button in the Spread area. You will then have to use the arrow keys to nudge individual labels until they are centered under their respective tick marks. Also select all of the labels and choose Layout > Alignment > Align Top Edges. For all of these static text fields, use the NSTextField Info palette to uncheck the Editable and Selectable checkboxes in the Options area, and check the Enabled checkbox, as

necessary.

c. Drag an NSTextBox icon from the Views palette and position it between the "Speed Limiter:" and "mph" labels. Select the text field and type **`155.0`** in it to serve as a guide to sizing and positioning the text field so that the largest value fits and the text lines up; then type **`75.0`** as the default value. With the text field selected, click the right alignment button in the NSTextField Info palette.

d. Select all the new items, then choose Layout > Group in Box, and give the box no title and no border. Now the new items can be dragged as a group without selecting all of them. This is especially important given how much work you did to line up the tick mark labels.

e. Drag to position the box and its contents so that it complies with human interface guidelines and is legible. Use the arrow keys to center the text and text field above the slider.

2. Use Project Builder to write the code required to make the new slider and text field work.

a. **User control outlet variable and accessors**. In the header file `MyWindowController.h`, declare outlet variables after the existing variable declarations, as follows:

```
// Speed slider
IBOutlet NSSlider *speedSlider;
IBOutlet NSTextField *speedTextField;
```

Still in `MyWindowController.h`, also declare accessor methods for the outlets at the end of the Accessor methods and conveniences section, as follows:

```
// Speed slider
- (NSSlider *)speedSlider;
- (NSTextField *)speedTextField;
```

Turn to the source file `MyWindowController.m` and define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Speed slider

- (NSSlider *)speedSlider {
    return speedSlider;
}

- (NSTextField *)speedTextField {
    return speedTextField;
}
```

b. **Data variable and accessors**. In the header file `MySettings.h`, declare a new variable after the existing variable declarations, as shown below. You need only the one data variable, because both of the new user controls represent the same value.

```
// Speed
float speedValue;
```

In `MySettings.h`, also declare the corresponding accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Speed

- (void) setSpeedValue:(float)value;
- (float) speedValue;
```

In the source file `MySettings.m`, define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Speed

- (void) setSpeedValue:(float)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setSpeedValue:speedValue];
    speedValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRSpeedValueChangedNotification
object:self];
}

- (float)speedValue {
    return speedValue;
```

```
}
```

c. **Notification variable**. In the header file `MySettings.h`, at the bottom of the file, declare the notification variable, as follows:

```
// Speed
extern NSString *VRSpeedValueChangedNotification;
```

In the source file `MySettings.m`, near the top of the file, define the notification variable, as follows:

```
// Speed
NSString *VRSpeedValueChangedNotification =
@"SpeedValue Changed Notification";
```

d. **GUI update method**. In the header file `MyWindowController.h`, declare user interface update methods at the end of the Specific view updaters section, as follows:

```
// Speed
- (void)updateSpeedSlider:(NSNotification
*)notification;
- (void)updateSpeedTextField:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define the specific update methods at the end of the Specific view updaters section, as follows:

```
// Speed

- (void)updateSpeedSlider:(NSNotification
*)notification {
    [self updateSlider:[self speedSlider]
setting:[[self mySettings] speedValue]];
}

- (void)updateSpeedTextField:(NSNotification
*)notification {
    [self updateTextField:[self speedTextField]
setting:[NSString localizedStringWithFormat:@"%f",
[[self mySettings] speedValue]]];
}
```

Here, in the `updateSpeedTextField:` method, you see another use of the NSString `localizedStringWithFormat:` class method. It is used here in a typical setting, where all that is needed is a number in string form, without any other text to be concatenated. You have to convert the floating point data value to a string, because the `updateTextField:` method, which you will write next, takes an NSString parameter.

There is a subtle but important point to be made here about these update methods. For the first time, you are arranging to update a text field, which presents many more complexities than a simple button or slider. If several windows are open in the Vermont Recipes application at once, for example, it is possible that a user will start to edit the text field in one window, then switch to another window and begin editing the very same text field in it, leaving the edit in the first window pending with an active cursor. Unlike with buttons or sliders, a user can end up with many text fields, each in a different document window, in "suspended animation" at once. Without careful coding, when the user attempts to commit one of the fields, these update methods might not know in which window the field was committed. Fortunately, in *Recipe 1, Step 5.3*, you arranged for the notification that triggers these update methods to be sent only to observers who registered for notifications containing the specific MySettings object that is associated with this particular document's window. Otherwise, the notification would be received and acted upon by all instances of this text field in every open window, causing them all to update at once. It would look to your user as though the text field in the other windows had aborted their pending edits and reverted to their previous data values, each on its own initiative. Because you have ensured that the notification goes only to the particular window controller that is associated with the specific settings object that posted the notification, you have avoided this puzzling and hard to debug behavior.

As you see, a new generic updater is required for textfields, which will receive string values. In the header file `MyWindowController.h`, declare a user interface update method at the end of the Generic view updaters section, as follows:

```
- (void)updateTextField:(NSTextField *)control
setting:(NSString *)value;
```

In the source file `MyWindowController.m`, define this update method at the end of the Generic view updaters section, as follows:

```
- (void)updateTextField:(NSTextField *)control
setting:(NSString *)value {
    if (value != [control stringValue]) {
        [control setStringValue:value];
    }
}
```

e. **Notification observer**. Register the window controller as an observer of the notification that will trigger the updater method. Two methods must be registered for the same notification, one to update the slider and the other to update the textfield. Insert the following statements in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the existing registrations:

```
// Speed
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateSpeedSlider:)
name:VRSpeedValueChangedNotification object:[self
mySettings]];
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateSpeedTextField:)
name:VRSpeedValueChangedNotification object:[self
mySettings]];
```

f. **Action method**. In the header file `MyWindowController.h`, at the end of the Action methods section, declare two action methods as shown below. Two action methods are needed because, as you will see in instruction 3, below, when either of the two controls is operated by the user, the other must be updated automatically.

```
// Speed
- (IBAction)speedSliderAction:(id)sender;
- (IBAction)speedTextFieldAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action methods at the end of the Action methods `section`, as shown below.

```
// Speed

- (IBAction)speedSliderAction:(id)sender {
    [[self mySettings] setSpeedValue:[sender
floatValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Speed
Limiter", @"Name of undo/redo menu item after Speed
slider was set")];
}

- (IBAction)speedTextFieldAction:(id)sender {
    [[self mySettings] setSpeedValue:[sender
floatValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Speed
Limiter", @"Name of undo/redo menu item after Speed
text field was set")];
}
```

g. **Localizable.strings**. Update the `Localizable.strings` file by adding "Set Speed Limiter."

h. **Initialization**. Initialize the Speed data variable to its minimum value, 75.0, using this statement at the end of the Default settings values section near the top of `MySettings.m`:

```
[self setSpeedValue:75.0];
```

i. **Data storage**. In the source file `MySettings.m`, define the following key in the Keys and values for dictionary section:

```
// Speed
static NSString *speedValueKey = @"SpeedValue";
```

Immediately after that, add these lines at the end of the `convertToDictionary:` method:

```
// Speed
[dictionary setObject:[NSString
stringWithFormat:@"%f", [self speedValue]]
forKey:speedValueKey];
```

The `stringWithFormat:` method is used here, rather than `localizedStringWithFormat:`, because this method uses a string format only to store the data as a byte stream. There is no need to take time to store a localized string to disk, anyway, since it would just be relocalized when it is retrieved and displayed.

Then add these lines near the end of the `restoreFromDictionary:` method, before the call to `[[self undoManager] enableUndoRegistration]`:

```
// Speed
[self setSpeedValue:(float)[[dictionary
objectForKey:speedValueKey] floatValue]];
```

j. **GUI update method invocation**. Finally, in `MyWindowController.m`, add the following calls at the end of the `updateWindow` method:

```
// Speed
[self updateSpeedSlider:nil];
[self updateSpeedTextField:nil];
```

3. Cocoa provides built-in methods that allow one user control to take its value directly from another control, without putting you through any convoluted routines to update the control from the value of a data variable. There need not even be a data variable associated with the two controls that are tied together.

   Here, however, you don't need to use these built-in methods, because the notification-based updating scheme you have adopted keeps the slider and the editable text field linked automatically. When the user sets a value on either of these two controls, its action method sets the shared data value in MySettings by calling the appropriate accessor method. The accessor method then posts a notification signaling that the data value has changed. Finally, since both the slider's and the text field's update methods have been registered to receive that notification, both controls are updated.

   You will see an example of Cocoa's automatic control linking methods in *Step 3*.

4. If you were to connect the outlets and actions in Interface Builder now and use these new controls, you would find that the numbers in the new text field are not formatted appropriately. When you set an exact integer value, no decimal point appears (that is, 75.0 displays as "75"), and when you set a non-integer value, many digits to the right of the decimal point may appear. To cure this cosmetic problem, you will use one of the built-in formatters supplied with Cocoa, NSNumberFormatter, using Interface Builder. (The old NSCell `setEntryType:` method familiar to NextStep programmers is now deprecated in favor of NSFormatter and its subclasses.)

   a. In Interface Builder, select the Data Views palette. It's the button with the word "Text"

appearing twice, third from the right in a fresh installation of Interface Builder.

b. Drag the icon with a dollar sign from the palette to the `MyDocument.nib` window. In the `MyDocument.nib` window, rename it **`SpeedFormatter`**. If your application were to need multiple text fields, all having the same formatting attributes that you are about to give this one, you could reuse the new SpeedFormatter formatter object to keep memory requirements down. For text fields with different formatting requirements, you would add more formatter objects.

c. Select the SpeedFormatter icon in the `MyDocument.nib` window. The NSNumberFormatter Info palette appears. Uncheck the Add 1000 Separators option and check the Localize option. (You have to turn off the Add 1000 Separators option before you can set the format of positive, zero and negative values that don't include separators.)

d. In the Info palette, for a Positive format type **`##0.0`**. You won't allow zero or negative values, but you can set them, as well, if you like, to **`0.0`** and **`-##0.0`**, respectively, in case you later extend the range to encompass zero or negative values.

e. Set the Minimum and Maximum values to 75 and 155, respectively.

f. Control-drag from the text field to the new SpeedFormatter formatter icon, select formatter in the Connections pane of the NSTextField Info palette, and click the Connect button.

If you were to connect the outlets and actions in Interface Builder again and use these new controls with the formatter, you would find that the formatting is just what you want, and that the minimum and maximum values you set in the formatter are enforced by a beep when you attempt to enter an out-of-range or non-numeric value in the text box. You will live with the beep for now, but you will add a powerful sheet in *Recipe 4* to give your users better feedback and increased options.

5. Inform the nib file of the new outlet and action you have created in the code files, then connect them to the new slider.

a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

b. Select the Instances tab and Control-drag from the File's Owner icon to the new slider and click its outlet name, then click the Connect button in the Connections pane of the File's Owner Info palette. Repeat this procedure with the new text field, connecting its outlet.

c. Control drag from the new slider to the File's Owner icon and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the NSSlider Info palette, then click the Connect button. Repeat this procedure with the new text field to connect its action.

Compile and run the application to test the slider and the text field. Be sure to explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly.

Notice how the value in the text field changes continuously as you drag the slider back and forth. Also, type floating point numbers into the text field and watch the slider snap to the corresponding settings once you hit the Return or Enter key.

Finally, type an out-of-range value or the word "fast" into the text field and hit Return. You will hear a beep telling you that you made an error, and the text field will still have an active cursor inviting you to edit the erroneous entry.

If you test thoroughly, however, you will discover two problems with the text field: It allows you to enter a blank value or **`0.0`**, then displays it as an unformatted, out-of-range floating point zero value. And it allows you to use your custom navigation buttons to switch to another tab view item even if an invalid entry is still pending (but the system prevents you from switching by clicking on a tab). You will fix these two problems and enhance the text field with a sheet in *Recipe 4, Step 3*. First, however, you will create one more slider in the next Step.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

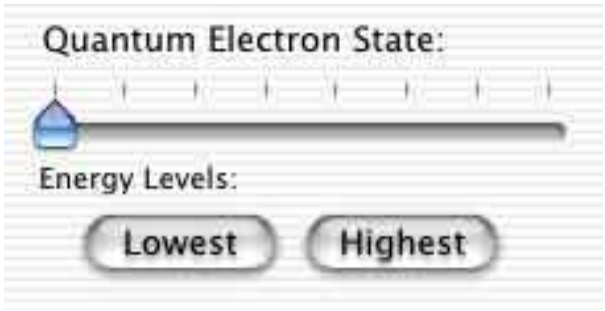
August 1, 2001 - 6:00 PM

Recipe 3: User controls—Sliders

Step 3: A slider with push buttons

- Highlights:
  - Setting a slider from push buttons
  - Setting a static text field from a slider's value continuously as the slider is dragged

In this Step, you will create a horizontal slider with tick marks in a range, as in *Step 2*, but the floating-point values it will return will be constrained to unit intervals. The thumb of the slider will jump from tick mark to tick mark in discrete increments as you drag it. The slider will let you set the quantum energy levels of electrons, which can only assume integer values (at least for purposes of this Step).



The slider in this Step has two associated push buttons. Clicking one or the other resets the slider to its minimum or maximum value.

A static text field is associated with this slider, as a read-out of the numeric value of the slider. Unlike the text field in the previous Step, this text field cannot be edited. The slider and the buttons are the only means to enter values. However, like the text field in the previous step, this static field's value is updated continuously as the slider is dragged, using a shortcut method in NSControl for linking user controls.

1. Use Interface Builder to create a new horizontal slider with a title, tick marks and labels, along with two push buttons and a label. When you are done with this instruction 1, the group should look like that shown in *Screenshot 3-4*.

a. With the Sliders tab view item selected, drag the topmost horizontal slider from the More Views palette onto the Sliders pane and place it in the lower right area of the pane. In the NSSlider Info palette, type **`0.0`** as the Minimum Value and **`7.0`** as the Maximum Value (the current value should automatically assume a value within that range, but type **`0.0`** for the Current Value, if necessary); check the Continuous checkbox; set the Number of Markers to 8 with Position Above; and check Marker Values Only. Note that "Continuous" in this Info palette refers to the frequency with which the control's action method is called; it is unrelated to the "Marker Values Only" setting, which determines whether the thumb may be positioned continuously along the length of the slider or only at tick marks.

b. Drag Message Text from the Views palette to the Sliders pane, placing it above the new slider and naming it **`Quantum Electron State`**; choose Layout > Size to Fit to make all the text visible. Drag Informational Text to the Sliders pane, placing it below the new slider, then type **`Energy Level:`** with a trailing colon, choose Layout > Size to Fit, and position it beneath the leftmost tick mark. Uncheck the Editable and Selectable checkboxes in the Options area, and check the Enabled checkbox, as necessary for both of these static text fields.

c. Now drag Informational Text to the Sliders pane again, placing it to the right of the Energy Level text field. Type the numeral **`0`** to match the default setting of the slider, and choose Layout > Size to Fit. This text field, though not editable or scrollable, will contain a value that varies depending on the user's use of the slider. The text field should therefore be marked Selectable in the Info palette in order to permit the user to select and copy its contents to the clipboard for pasting elsewhere.

d. Drag standard push buttons from the Views palette twice and position them side by side under the Energy Levels text field. Name the buttons **`Lowest`** and **`Highest`**, respectively.

e. Select all the new items, then choose Layout > Group in Box, and give the box no title and no border. Now the new items can be dragged as a group without selecting all of them.

f. Position the group so that it complies with human interface guidelines and is legible.

2. Use Project Builder to write the code required to make the new controls work.

a. **User control outlet variable and accessors**. In `MyWindowController.h`, declare outlet variables after the existing variable declarations, as follows:

```
// Quantum slider
IBOutlet NSSlider *quantumSlider;
IBOutlet NSButton *quantumButton1;
IBOutlet NSButton *quantumButton2;
```

In `MyWindowController.h`, also declare accessor methods for the outlets near the end of the Accessor methods and conveniences section, as follows:

```
// Quantum slider
- (NSSlider *)quantumSlider;
- (NSButton *)quantumButton1;
- (NSButton *)quantumButton2;
```

In `MyWindowController.m`, define the accessor methods at the end of the `Accessor methods and conveniences` section, as follows:

```
// Quantum slider

- (NSSlider *)quantumSlider {
    return quantumSlider;
}

- (NSButton *)quantumButton1 {
    return quantumButton1;
}

- (NSButton *)quantumButton2 {
    return quantumButton2;
}
```

b. **Data variable and accessors**. In `MySettings.h`, declare a new variable after the existing variable declarations, as follows:

```
// Quantum
float quantumValue;
```

In `MySettings.h`, also declare the corresponding accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Quantum

- (void)setQuantumValue:(float)value;
- (float)quantumValue;
```

In `MySettings.m`, define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Quantum

- (void)setQuantumValue:(float)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setQuantumValue:quantumValue];
    quantumValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRQuantumValueChangedNotification
object:self];
}

- (float)quantumValue {
    return quantumValue;
}
```

c. **Notification variable**. In `MySettings.h`, at the bottom of the file, declare the notification variable, as follows:

```
// Quantum
extern NSString *VRQuantumValueChangedNotification;
```

In `MySettings.m`, near the top of the file, define the notification variable, as follows:

```
// Quantum
NSString *VRQuantumValueChangedNotification =
@"QuantumValue Changed Notification";
```

d. **GUI update method**. In `MyWindowController.h`, declare a user interface update method at the end of the Specific view updaters section, as follows:

```
// Quantum
- (void)updateQuantumSlider:(NSNotification
*)notification;
```

In `MyWindowController.m`, define the specific update method at the end of the Specific view updaters section, as follows:

```
// Quantum

- (void)updateQuantumSlider:(NSNotification
*)notification {
    [self updateSlider:[self quantumSlider]
setting:[[self mySettings] quantumValue]];
}
```

Notice that the existing `updateSlider:` generic update method will be reused.

e. **Notification observer**. Register the window controller as an observer of the notification that will trigger the updater method, at the end of the `registerNotificationObservers` method in `MyWindowController.m`, after the existing registrations:

```
// Quantum
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateQuantumSlider:)
name:VRQuantumValueChangedNotification object:[self
mySettings]];
```

f. **Action method**. In `MyWindowController.h`, at the end of the Action methods section, declare three action methods as shown below. Action methods are needed for the buttons because they will update the slider.

```
// Quantum
- (IBAction)quantumSliderAction:(id)sender;
- (IBAction)quantumButton1Action:(id)sender;
- (IBAction)quantumButton2Action:(id)sender;
```

In `MyWindowController.m`, define the action methods at the end of the Action methods section, as shown below.

```
// Quantum

- (IBAction)quantumSliderAction:(id)sender {
    [[self mySettings] setQuantumValue:[sender
floatValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Quantum
Electron State", @"Name of undo/redo menu item
after Quantum slider was set")];
}

- (IBAction)quantumButton1Action:(id)sender {
    [[self mySettings] setQuantumValue:[[self
quantumSlider] minValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Lowest
Quantum Electron State", @"Name of undo/redo menu
item after Quantum button 1 was set")];
}

- (IBAction)quantumButton2Action:(id)sender {
    [[self mySettings] setQuantumValue:[[self
quantumSlider] maxValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Highest
Quantum Electron State", @"Name of undo/redo menu
item after Quantum button 2 was set")];
}
```

Notice that, because you decided early on to set the action name in the action method rather than in the data variable's accessor method, you are able to use different, more informative, names for the Undo and Redo menu items, depending on exactly what action the user performed.

g. **Localizable.strings**. Update the `Localizable.strings` file by adding "Set Quantum Electron State."

h. **Initialization**. No initialization is required to set the slider to 0.

i. **Data storage**. In `MySettings.m`, define the following key in the Keys and values for dictionary section:

```
// Quantum
static NSString *quantumValueKey = @"QuantumValue";
```

Immediately after that, add these lines at the end of the `convertToDictionary:` method:

```
// Quantum
[dictionary setObject:[NSString
stringWithFormat:@"%f", [self quantumValue]]
forKey:quantumValueKey];
```

Add these lines at the end of the `restoreFromDictionary:` method:

```
// Quantum
[self setQuantumValue:(float)[[dictionary
objectForKey:quantumValueKey] floatValue]];
```

j. **GUI update method invocation**. In `MyWindowController.m`, add the following call at the end of the `updateWindow` method:

```
// Quantum
[self updateQuantumSlider:nil]
```

3. In instruction 1., above, you used Interface Builder to insert a non-editable text field after "Energy Levels:" to hold a numerical representation of the value of the slider. Now you need to add code to ensure that the contents of this text field are updated continuously as the slider is dragged back and forth. In *Step 2*, you linked a slider and a text field as a fortuitous side-effect of the notification-based updating scheme used in the Vermont Recipes application. There, the link depended on the fact that the slider and the text field shared a single data variable, and action and updater methods were required for both controls so that both could be used to set the data value.

   Here, the text field is not editable, and it therefore has no action or update method and is not associated with a data value. You need another means to link it to the slider. It happens that NSControl contains a small set of methods designed precisely for this situation. Here, you will use NSControl's `takeIntValueFrom:` method to tell the text field to take its value directly from the value of the slider. Since you set the slider's Continuous Option in instruction 1., above, the updating of both the slider and the text field will occur continuously as the slider is dragged because the slider's action method will be invoked continuously.

   First, in `MyWindowController.h`, add this declaration of an `IBOutlet` variable for the text field, at the end of the Quantum slider variables section:

```
IBOutlet NSTextField *quantumTextField;
```

Also in `MyWindowController.h`, declare the accessor method at the end of the Accessor methods and conveniences section:

```
- (NSTextField *)quantumTextField;
```

Define the accessor method in `MyWindowController.m` at the end of the Accessor methods and conveniences section:

```
- (NSTextField *)quantumTextField {
    return quantumTextField;
}
```

Finally, add the following statement at the end of the `updateQuantumSlider:` method in `MyWindowController.m`:

```
[[self quantumTextField] takeIntValueFrom:[self
quantumSlider]];
```

This statement goes in the slider's update method, not its action method. If it were placed in the action method, it would be invoked only when the user clicks on the slider. But you want the text field to update also when the document is created or opened, and after a Redo or Undo command, and when the document is reverted to its saved state. This statement must therefore go in the slider's update method, which will be invoked every time the slider's data value changes, no matter how the user initiated the change. This is why, in *Step 1*, you went to the trouble of invoking updaters for specific user controls in the `updateInterface` method; namely, so that methods that are specific to an individual control, such as the `takeIntValueFrom:` method in the Quantum slider control, will be invoked even when the document window is first opened and when the document reverts to its saved state. If you had instead invoked the generic slider update method in `updateInterface`, the `takeIntValueFrom:` method would not have been called in these circumstances.

4. Inform the nib file of the new outlets and actions you have created in the code files, then connect them to the new slider and push buttons.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse

button.

b. Select the Instances tab and Control-drag from the File's Owner icon to each of the new controls in turn and click its outlet name, then click the Connect button in the Connections pane of the File's Owner Info palette.

c. Control drag from each of the new controls to the File's Owner icon in turn and click the target in the left pane and the appropriate action in the right pane of the Outlets section of the Connections pane of the Info palette, then click the Connect button.

Compile and run the application to test the slider and the two buttons. As usual, be sure to explore how undo and redo work, and make sure changes can be saved to disk, restored, and reverted properly. Notice how the value in the text field changes instantly to its lowest or highest value when you press the appropriate push button. Be sure to try copying the value in the text field and pasting it into, say, TextEdit.

You are now done with *Recipe 3*. In *Recipe 4*, you will take a short detour from user controls to learn how to create more complex sheets, and in *Recipe 5* you will learn how to implement on-the-fly input filtering as a user types characters into a field, as a prelude to creating more complex text fields. In *Recipe 6*, you will wrap up your exploration of user controls by implementing a variety of text fields and other controls, such as forms, that contain text fields.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 4: User controls—Text fields (sheets)

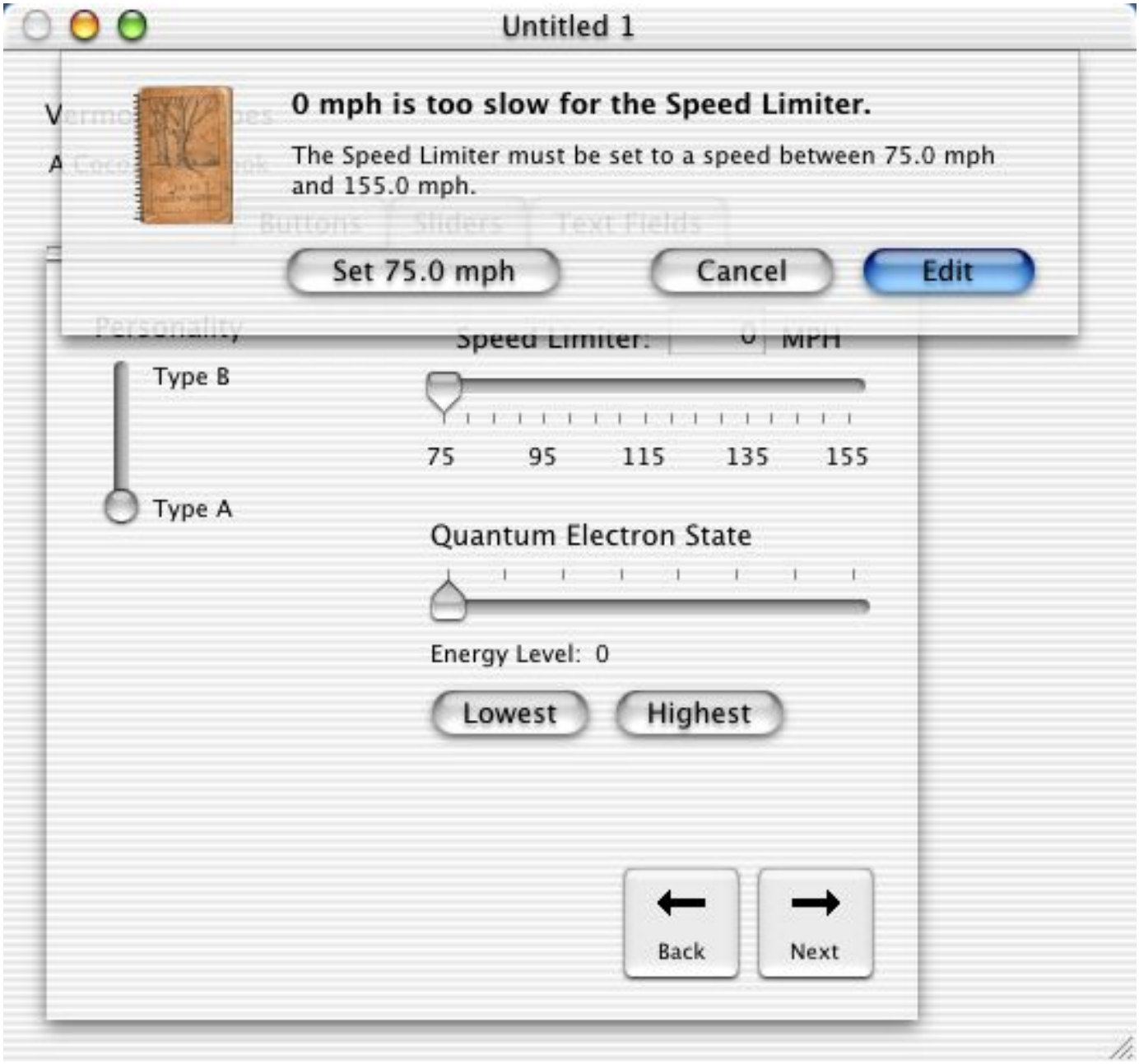
Download the project files for Recipe 4 as a disk image and install them

Download a pdf version of Recipe 4

This Recipe is the third in a series of Recipes dealing with user controls. In *Recipes 2* and *3*, you implemented buttons of various kinds and sliders. You were also briefly introduced to text fields and two features of Cocoa that are particularly useful with text fields: sheets and formatters. Now, in the next three Recipes, you will learn more about text fields, sheets and formatters. In this *Recipe 4*, you will get a thorough grounding in interactive sheets. In *Recipe 5*, you will learn more about custom formatters. And in *Recipe 6*, you will learn more about text fields proper, including their use in more complex user controls such as forms, tables and outlines.

Before turning to *Step 1*, you should prepare your project files for *Recipe 4*. Follow the same procedures you followed in the introduction to *Recipe 3*, which were a small subset of those outlined in *Recipe 2, Step 1*.

Screenshot 4-1: The Vermont Recipes 4 application



Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe04/recipe04.html

Introduction > Contents > Recipe 4 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 4: User controls—Text fields (sheets)

Step 1: A better way to save documents

- Highlights:
  - Saving a document in XML format
  - Automatically preserving old versions as backup files

Before getting into the main subject of *Recipe 4*, sheets, you will take a brief detour to discuss a better way of doing something that you have already learned how to do. We are starting to find it helpful to use the first or second Step of a Recipe to revisit something previously taught, as we did once before in *Recipe 2, Step 2*, where you revised how the Vermont Recipes application handles data initialization. This will probably become a habit, since taking one step back after two or three steps forward seems to be a common way to learn new material.

In *Recipe 1, Step 4.2*, you learned one way to save a document's data to persistent storage and retrieve it. There, you used a dictionary object as a vehicle to save the data in Cocoa's old property list format, a text format that consists of a list of key-value pairs, each separated by an equals sign. Now, in this Step, you will learn how to save a document in XML format. Calling this "a better way" to save data is a bit of an exaggeration, as the way you did it originally is a perfectly viable method for saving data in appropriate circumstances. But XML has many uses, and this is a good opportunity to learn one utterly painless way to save your data in XML format.

XML (Extensible Markup Language, a W3C Recommendation) is storming the world because it is a generalized computer-readable text format for structured data. The old Cocoa property list format is more human-readable, because your eyes don't have to sift through a lot of XML tags in angle brackets, some of which don't have obvious meanings. But XML is a universal format for storing computer data in text form,

amenable to automated parsing by XML utilities and easy integration with data obtained from other sources. Once a DTD (document type definition) for a particular XML data set is publicly available, as the Cocoa property list DTD is (on your computer at */System/Library/DTDs/PropertyList.dtd*), any XML utility can read and manipulate its data. For example, in Mac OS X, Late Night Software's free XML Tools scripting addition can be used to parse data files saved by your application in XML format and extract individual items using AppleScript, thus making your application a better citizen of the Mac OS X world even if it is not itself scriptable. If you're curious how this might work, examine the Lookup Vermont Recipes Key script, which uses XML Tools in a script to look up a value in a Vermont Recipes data file, given a valid key. (This script is specially designed to understand the specific Vermont Recipes XML data format used in this Step; it is not a generalized XML parser.) It is likely that Mac OS X will acquire even more support for the XML format as time goes by. Your application may gain a wider market if potential customers know that its data will be accessible in this fashion. (Of course, you will sacrifice the economic advantages that a proprietary data format gives to a product that already enjoys market dominance in its class; even if you are lucky enough to have a product in this category, XML might be a good data interchange format for your application.)

Cocoa's NSDocument class contains facilities allowing a document-based application to save data to persistent storage by overriding built-in methods at various levels. In *Recipe 1, Step 4.2*, you took what was an easy path, overriding the `dataRepresentationOfType:` method that the Cocoa Document-based Application template placed in your MyDocument class for you when you first created the Vermont Recipes 1 project files. However, Cocoa allows you to override its `writeToFile:OfType:` method, instead, as one data storage alternative. The documentation describes the former as a "data-based primitive" and the latter as a "location-based primitive." You can use either, and in this step you will choose to use the location-based primitive.

Your `writeToFile:OfType:` method will simply invoke a method of the same name declared in Cocoa's NSDictionary class. Methods with this name exist in several Cocoa classes for just this purpose, namely, to let you call them in your document class's override of the NSDocument `writeToFile:OfType:` method. For example, NSString contains a version of this method. The NSDictionary implementation of this method is just what you want here, because the AppKit Release Notes for Mac OS X 10.0 disclose that, with this release, NSDictionary now writes its data in XML format by default. (NSArray also contains a method with the same name, which also now writes in XML format by default.) NSDocument, NSDictionary and other classes also contain `readFromFile:OfType:` methods which let you read XML files back into memory from persistent storage.

1. You don't want to throw away work already undertaken, in case this new technique doesn't pan out. So, for the time being, "comment out" the existing code in `MyDocument.m` that you will be replacing. In the Saving information to persistent storage subsection of the Persistent storage section, bracket the entire implementations of the `dataRepresentationOfType:` and `convertForStorage` methods with a pair of comment brackets, `/*` and `*/`. You must also comment out the declaration of `convertForStorage` in the header file, `MyDocument.h`. Keep the `setupDictionaryFromMemory` method, however, because you need it to set up your document's data in RAM as a temporary dictionary, which will then write itself to persistent

storage in XML format.

2. In `MyDocument.m`, add the following override method after the methods that you commented out:

```
- (BOOL)writeToFile:(NSString *)fileName
ofType:(NSString *)type {
    if ([type isEqualToString:myDocumentType]) {
        [[self undoManager] removeAllActions];
        return [[self setupDictionaryFromMemory]
writeToFile:fileName atomically:YES];
    } else {
        return NO;
    }
}
```

You needn't declare this method in `MyDocument.h`, since it is an override method.

The key statement is in the middle of the method: `return [[self setupDictionaryFromMemory] writeToFile:fileName atomically:YES]`. This simply tells the dictionary object that you created in `setupDictionaryFromMemory` to write itself to persistent storage, which it now, in Mac OS X 10.0, does in XML format. It automatically uses UTF-8 encoding.

You will recognize the rest of the code in this method as having been stolen from your old `dataRepresentationOfType:` override method, which used it to make sure the document being written to persistent storage is of the right type and, if so, to remove any pending undo and redo actions from the queue. There, you returned `nil` if the document was of the wrong type, because the `dataRepresentationOfType:` method is supposed to return an object; here, you return `NO`, because the `writeToFile:OfType:` method is supposed to return a Boolean value.

The `atomically` parameter specifies whether the file should be fully written to persistent storage before the old version of the file is deleted. You normally want to pass `YES` to this parameter to ensure against loss of data due to power failures and the like.

One final comment: The data-storage technique that you just abandoned invoked a built-in NSDictionary method, `description`, to generate the text for the key-value pairs that you then saved to persistent storage. You might wonder why the `description` method hasn't been updated in Mac OS X 10.0, as `writeToFile:OfType:` was, to provide an XML rendition of the data instead of a list of equated key-value pairs. The answer is that virtually every Cocoa class implements a `description` method whose principal purpose is to assist you in debugging your applications. The debugger used in Project Builder does not provide a GUI for reading the current

values of the instance variables declared in an object, but this is information that would be very useful to you when debugging. What you do is to use the "print object" or "po" command in the debugger, which calls the object's `description` method, if it has one, and shows you its instance variables' values. Because the point of the `description` method is to provide a readily human-readable text, it would not make sense to have it return an XML rendition. (This is not to say that Cocoa shouldn't provide a method in many objects to return their values in XML format; this could be called something like `descriptionInXMLFormat`. But this is beyond the scope of *Vermont Recipes*.)

3. If you were to compile and run the application at this point, you might be surprised to discover that you could stop here, if you wanted to. Try it. Once the initial, untitled Vermont Recipes document is visible on the screen, save it as is and close its window. Then, in the Finder, drag the document's icon onto the TextEdit application icon to see what it looks like. Sure enough, you see the telltale tag-filled text of an XML document, complete with elements telling you what version of the XML standard is used and where the property list DTD can be found. That isn't the surprise, though—after all, you expected this to work, didn't you? The surprise comes when you then return to the Vermont Recipes application and try to open the document you just saved. It works! How could this be? You haven't yet revised your code to *read* XML documents, but only to *write* XML documents.

   The explanation lies in the NSString `propertyList` method that you used in your `setupDictionaryFromStorage:` method in *Recipe 1, Step 4.2.2* to convert the file's data to dictionary form after it is read back in from persistent storage. It already knows how to interpret XML data as well as old-style property list data.

4. However, there is value in symmetry, and the methods you wrote previously to read the file from persistent storage were not as efficient as they might be, so you will now revise the methods that read a Vermont Recipes document to correspond to the pattern you used to save it to persistent storage in instruction 2., above.

   For the time being, "comment out" the existing code in `MyDocument.m` that you will be replacing. In the Loading information from persistent storage subsection of the Persistent storage section, bracket the entire implementations of the `loadDataRepresentation:ofType:`, `restoreFromStorage:` and `setupDictionaryFromStorage:` methods with a pair of comment delimiters, `/*` and `*/`. You must also comment out the declarations of `restoreFromStorage:` and `setupDictionaryFromStorage:` in the header file, `MyDocument.h`.

5. In `MyDocument.m`, add the following override method after the methods that you commented out:

```
- (BOOL)readFromFile:(NSString *)fileName
ofType:(NSString *)type {
    NSDictionary *dictionary;
    if ([type isEqualToString:myDocumentType]) {
        dictionary = [NSDictionary
dictionaryWithContentsOfFile:fileName];
        [self restoreFromStorage:dictionary];
        return (dictionary != nil);
    } else {
        return NO;
    }
}
```

You needn't declare this method in `MyDocument.h`, since it is an override method.

Notice that this method retrieves the data from persistent storage using NSDictionary's `dictionaryWithContentsOfFile:` class method. You could have used this method before, as well, instead of taking the more circuitous route of reading the file into a string with NSData's `initWithData:` method and then converting the string to a dictionary using NSString's `propertylist` method. In general, many of Cocoa's Foundation classes, not just NSData, include methods for reading data directly into an instance of the class from persistent storage, just as many of them include methods for writing their data directly to persistent storage.

If you're paying close attention, you wonder why you were asked to comment out the `restoreFromStorage:` method in instruction 4., above, since it is called in this code snippet. The answer is that you need to redefine it, which you will do in the next instruction.

6. The application needs a way to use the data in the dictionary to update its graphical user interface. You commented out the `restoreFromStorage:` method, where you formerly performed this task. You can't use that method as is, because it takes an NSData parameter. Instead, you will create a similar method with the same name that takes the new dictionary object as a parameter.

   In `MyDocument.m`, after your new `readFromFile:ofType:` override method, add the following:

```
- (void)restoreFromStorage:(NSDictionary *)dictionary {
    if (dictionary) {
        [[self mySettings]
restoreFromDictionary:dictionary];
    }
}
```

This method does exactly what the old method of the same name did, namely, tell the model object to tell the window controller to update the user interface with the new data. The only difference is that it now receives the dictionary from NSDictionary's `dictionaryWithContentsOfFile:` method instead of creating it from an NSData object. If you create additional model objects in the future, you have only to call their `restoreFromDictionary:` methods here, immediately after the call to the MySettings version of the method.

The `if (dictionary)`... test checks to see whether the incoming dictionary object is nil, using shorthand, as is common in Cocoa code, for `if (dictionary != nil)`....

Don't forget to add the declaration of this method in `MyDocument.h`:

```
- (void)restoreFromStorage:(NSDictionary *)dictionary;
```

7. Before deleting the code you commented out, compile and run the application, and write a file to persistent storage and read it back to make sure these routines are working. Since the File > Revert command also reads a file from persistent storage, test that, as well. If all is well, delete the code that you commented out from `MyDocument.h` and `MyDocument.m`.

8. As long as you're improving the application's data storage routines, why not arrange for automatic maintenance of backup files? It is as simple as overriding NSDocument's `keepBackupFile` method to return `YES` instead of the default `NO`. Add the following to `MyDocument.m` at the top of the Saving information to persistent data section:

```
- (BOOL)keepBackupFile {
    return YES;
}
```

Now, every time you choose File > Save to save recent changes to a Vermont Recipes document, the old version will be retained, instead of being deleted after the new version is successfully written to disk. The old version, now serving as a backup, retains the original name except for a tilde ("~") appended at the end of the file name, just before the file extension. Subsequent saves will delete the last backup file and replace it with the new backup file, so you will never have more than one backup file on your disk at a time. This is an awfully limited backup scheme, but it is a lot better than none at all, giving you a layer of protection that goes one step beyond the Revert command.

You have now substituted a more useful file format, XML, for the old-style property list format in which Vermont Recipes documents were originally saved. In the process, you have provided for a system of automatic backup files to protect your data against changes that don't work out. In the next Step, you will return to this Recipe's principal task, providing for interactive sheets.

Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe04/recipe04_step01.html

Vermont Recipes—A Cocoa Cookbook for Mac OS X

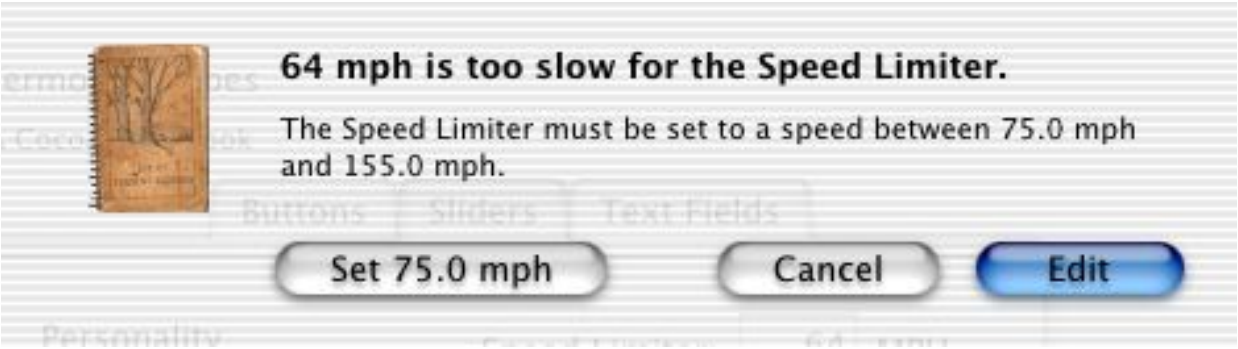
By Bill Cheeseman

August 1, 2001 - 6:00 PM

Recipe 4: User controls—Text fields (sheets)

Step 2: A complex, interactive document-modal sheet to deal with an invalid text field entry

- Highlights:
  - Moving a tab view item to a new position in a tab view
  - Using the `NSAlert()` function to catch configuration errors at run time
  - Internationalizing a sheet



In this *Step 2*, you will take up the challenge posed in *Recipe 3, Step 2*. There, you associated an NSNumberFormatter object with a text field, but you provided the user with nothing more than a beep when an attempt was made to enter an illegal value. Now, you will add code to the application to present a document-modal sheet when validating the text field, in order to explain to the user the nature of any data entry problem and to offer options to resolve it. You saw how to present a simple, one-button sheet in *Recipe 3, Step 1*, using a single statement. The sheets you will create in this Recipe involve a number of complexities, including recognition of the choice the user made among multiple buttons.

Text validation and complex sheets have been deferred to *Recipe 4* because they are often associated with text fields. The other controls you have encountered up to this point, buttons and sliders, have their data types and constraints built in, so to speak. Your users can't accidentally enter an illegal value into a checkbox by clicking on it, and they can't exceed a slider's maximum value by inadvertently dragging it over the top. Although formatters and other techniques can be used to prevent similar errors in text fields, a sheet is required if you wish to provide the user with an explanation and options.

Macintosh human interface guidelines have dictated, since as long ago as 1986, that text entered into text fields should be tested for validity when the user hits the Return, Enter or Tab key or clicks anywhere other than in the text field. Implicitly, the last part of this rule must be taken at something less than face value. It may be understood to require a validity check when the user clicks anywhere else *on certain enabled objects within the text field's window*. Clicks in many other locations do not require validating a pending text edit. For example, clicking in another window, including another application's window, or on the desktop must be allowed without constraint in the modern, non-modal Macintosh environment. Being able to work in several windows at once greatly increases the flexibility of Macintosh usage; for example, the user may need to copy some text in a second window to paste into the text field currently being edited. The user's switching to another window or another application only temporarily suspends the editing of the text field, pending the user's return to its window, at which time its cursor will again become active. Clicking within an inactive part of the window, such as its background, also normally does not require a validity check on the text field, but instead leaves it active and ready to accept typing. Similarly, clicking on various other menu items, buttons, sliders and the like need not trigger a validity check on the text field because, among other things, these other user controls might be used to alter application settings that affect what is happening in the text field, such as selecting another font or style for a selected word. Clicking on another text field within the window will, however, trigger a validity check on the first text field, since only one text field may have keyboard focus within a window at any one time. There are other situations,

also not covered explicitly by the human interface guidelines, where you will follow Cocoa's default behavior. For example, switching to another tab view item within a window requires validation of a pending edit, while any pending edits are simply discarded when closing a window or quitting the application.

You will therefore test the validity of a text field's pending entry, and present a sheet if the entry is illegal, in two situations: when the user attempts to enter a value by pressing the Return, Enter or Tab key or clicking in another text field in the same window, and when the user attempts to switch to another tab view item in the same window. You will not validate a pending edit when the user attempts to close the window or to quit, because Cocoa regards these events as evidence of the user's intent to abandon a pending edit.

1. Before dealing with text validation, it is time to fix a cosmetic issue with the application. When you originally created the document window's tab view with two tab view items, in *recipe 1, Step 2.2*, you named the first tab "Text Boxes" and the second tab "Buttons." Later, in *Recipe 3, Step 1*, you added a third tab named "Sliders." Although the order of the tabs from left to right thus suggests that Text Boxes come first in *Vermont Recipes*, you are only now starting to deal with them. You will therefore make the Text Boxes tab come third in order, instead of first, to reflect the order of these Recipes. While you're at it, you will rename this tab view item "Text Fields."

    a. Open `MyDocument.nib` in Interface Builder, if necessary.

    b. Double-click in the tab view to select it, then click once on the Text Boxes tab to select that tab view item.

    c. Press the right arrow key on the keyboard twice to move the Text Boxes tab to the right of the other two tabs.

    d. Double-click on the "Text Boxes" tab's title to select it for editing, and change it to **`Text Fields`**.

    If you were to compile and run the application now, you would find that the navigation buttons enable and disable themselves as expected even though a tab view item has been moved to a new position. The code you wrote for these buttons in *Recipe 2, Step 8* was sufficiently generalized to make this happen automatically.

    Remember, however, that the Next button was disabled in Interface Builder on the assumption that the application would be compiled with the last tab view item showing. If you compile it with another tab view item showing, the navigation buttons will not be appropriately enabled and disabled when the window is first created, until the user changes to another pane for the first time. In anticipation of the final setup, where the first, or leftmost, tab will be selected immediately after a new window is created, use Interface Builder's NSButton Info palette now to disable the Back button and enable the Next button, and remember to compile the application with the first tab view item selected before releasing the product publicly.

Since memory isn't necessarily reliable, especially on a large team project, you should also consider including something in the application's code to ensure that the active tab view item and the enabled or disabled state of the navigation buttons are properly configured in the nib file. You have been taking care of the initial state of the window in MyWindowController's `windowDidLoad` method. One possibility would be to insert code there to set the state of the tab view programmatically. For example, you might add these statements:

```
[[self myTabView] selectFirstTabViewItem:nil;
[[self backButton] setEnabled:NO];
[[self nextButton] setEnabled:YES];
```

This may not be the best approach, however. For one thing, this would slow the opening of the window unnecessarily, if only modestly, since the nib file can and should be configured to set the state of these features of the window. More importantly, calling the `selectFirstTabViewItem:` method here would automatically trigger any delegate method you may later implement to respond to the user's selection of panes in the tab view, and it might be hard to track down any resulting misbehavior when the window is opened.

Therefore, instead of inserting the code just shown, it is better to call upon Cocoa's `NSAssert()` function to generate a runtime exception if the nib file is improperly configured. `NSAssert()` is designed to provide runtime checks and reminders of just this sort during application development. When you finally build the application for public release, you will use Project Builder's deployment build style setting to optimize this code out so that it will not adversely affect performance. Add the following to the end of the `windowDidLoad` method in `MyWindowController.m`:

```
NSAssert((([[self myTabView] tabViewItemAtIndex:0]
tabState] == NSSelectedTab), @"First tab view item is
not selected.");
NSAssert((![[self backButton] isEnabled]), @"Back
button is not disabled.");
NSAssert(([[self nextButton] isEnabled]), @"Next button
is not enabled.");
```

2. You will start creating a comprehensive text validation system for your text field by using Project Builder to write the code required to associate a document-modal sheet with the Speed Limiter text field in the Sliders tab view item. Recall that the Speed Limiter slider has an associated text field, where the user can type in a value between 75 and 155 instead of dragging the slider. You provided all the code necessary to handle the data and the text field itself in *Recipe 3, Step 2*, and you already associated an NSNumberFormatter object with the field. All you have to do here to give the user more useful feedback than a beep is to add some methods to your project files relating to the new sheet and appoint a delegate or two, in order to alert your user when illegal out-of-range or string

values are entered in the Speed Limiter text field. While the principles will be relatively easy to comprehend (though more complex than any you have encountered to this point in Vermont Recipes), the implementation is convoluted, involving the use of callback routines. Additional apparent complexity is contributed by the need to keep the implementation of sheets fully internationalized.

See Screenshot 4-2 to see how one of these sheets will look. Once this *Step 2* is completed and the basic sheets are implemented, you will turn to several related issues in *Step 3* and following.

3. *Detect the invalid text entry.* The first issue is how to detect the user's attempt to enter an illegal value. You already associated an NSNumberFormatter object with the text field, and the formatter clearly knows when an error has occurred because it beeps and refuses to accept the entry when you press the Return, Enter or Tab key. But how does your application find out about the error? There does not appear to be any method in NSFormatter or NSNumberFormatter to do this.

   The answer lies in the `control:didFailToFormatString:errorDescription:` delegate method which the authors of the NSControl class thoughtfully provided as an option for your use. As is the case with all of the many delegate methods in Cocoa, all you have to do to take advantage of it is to implement the delegate method in one of your application's classes (being careful to conform it exactly to the delegate method declaration in NSControl), then use Interface Builder to appoint your class as the delegate of the particular control in which you're interested. Cocoa will automatically invoke your implementation of the delegate method every time the formatter object attached to the control detects an attempt to enter illegal data. Here, you will appoint MyWindowController as the delegate of the Speed Limiter text field and implement the delegate method in MyWindowController.

4. In `MyWindowController.m`, define the delegate method as shown below, at the end of the source file before `@end`. You do not need to declare this method in the header file because it is a delegate method declared and called within the NSControl object.

```
// Input validation and formatting methods

    // Formatter errors

- (BOOL)control:(NSControl *)control
didFailToFormatString:(NSString *)string
errorDescription:(NSString *)error {
    if (control == [self speedTextField]) {
        return [self
sheetForSpeedTextFieldFormatFailure:string
errorDescription:error];
    } else {
        return YES;
    }
```

```
}
```

Once implemented, this delegate method will be invoked by any formatted text field in the application that appoints MyWindowController as its delegate, whenever there is an attempted illegal data entry in that field. Therefore, the first thing the method must do is to identify which control invoked it in this case and act accordingly. If it was not the Speed Limiter text field, and if it was not any other control for which MyWindowController might be appointed as delegate, it returns `YES`. This allows such a control to accept the attempted data entry. Your delegate method will not necessarily include a test for every control that appoints MyWindowController as delegate, so the `YES` branch is required.

5. *Present the sheet.* In the event the data entry error was generated by the Speed Limiter text field, the delegate method calls the custom `sheetForSpeedTextFieldFormatFailure:errorDescription:` method, which you will now write, passing along the invalid string and the error description. As you add text fields to your application, you will have to add custom sheet methods for each of them on this model, and you will also have to add branches to the test in the `control:didFailToFormatString:errorDescription:` delegate method for each new text field.

   At the end of `MyWindowController.h`, just before `@end`, declare the method that will handle the Speed Limiter text field error as follows:

```
// Input validation and formatting methods

    // Formatter errors

        // Speed Limiter text field

- (BOOL) sheetForSpeedTextFieldFormatFailure:(NSString
*)string errorDescription:(NSString *)error;
```

   In `MyWindowController.m`, define it as shown below at the end of the file before `@end`. This is the longest and most verbose method you have yet written in *Vermont Recipes*. Its interesting features will be explained in depth below.

```
// Speed Limiter text field

- (BOOL) sheetForSpeedTextFieldFormatFailure:(NSString
*)string errorDescription:(NSString *)error {
    NSString *alertMessage;
    NSString *alternateButtonString;
    float proposedValue;
    NSString *proposedValueString;

    NSString *alertInformation = [NSString
localizedStringWithFormat:NSLocalizedString(@"The Speed
Limiter must be set to a speed between %1.1f mph and %1.1f
mph.", @"Informative text for alert posed by Speed Limiter
text field when invalid value is entered"), [[self
speedSlider] minValue], [[self speedSlider] maxValue]];
    NSString *defaultButtonString =
NSLocalizedString(@"Edit", @"Name of Edit button");
    NSString *otherButtonString =
NSLocalizedString(@"Cancel", @"Name of Cancel button");

    if ([error
isEqualToString:NSLocalizedStringFromTableInBundle(@"Fell
short of minimum", @"Formatter", [NSBundle
bundleForClass:[NSFormatter class]], @"Presented when user
value smaller than minimum")]) {
        proposedValue = [[self speedSlider] minValue];
        alertMessage = [NSString
stringWithFormat:NSLocalizedString(@"%@ mph is too slow for
the Speed Limiter.", @"Message text for alert posed by Speed
Limiter text field when value smaller than minimum is
entered"), string];
        alternateButtonString = [NSString
localizedStringWithFormat:NSLocalizedString(@"Set %1.1f
mph", @"Name of alternate button for alert posed by Speed
Limiter text field when value smaller than minimum is
entered"), proposedValue];

    } else if ([error
isEqualToString:NSLocalizedStringFromTableInBundle(@"Maximum
exceeded", @"Formatter", [NSBundle
bundleForClass:[NSFormatter class]], @"Presented when user
value larger than maximum")]) {
        proposedValue = [[self speedSlider] maxValue];
        alertMessage = [NSString
```

```
stringWithFormat:NSLocalizedString(@"%@ mph is too fast for
the Speed Limiter.", @"Message text for alert posed by Speed
Limiter text field when value larger than maximum is
entered"), string];
        alternateButtonString = [NSString
localizedStringWithFormat:NSLocalizedString(@"Set %1.1f
mph", @"Name of alternate button for alert posed by Speed
Limiter text field when value larger than maximum is
entered"), proposedValue];

    } else if ([error
isEqualToString:NSLocalizedStringFromTableInBundle(@"Invalid
number", @"Formatter", [NSBundle bundleForClass:[NSFormatter
class]], @"Presented when user typed illegal characters --
No valid object")]) {
        alertMessage = [NSString
stringWithFormat:NSLocalizedString(@"“%@” is not a valid
entry for the Speed Limiter.", @"Message text for alert
posed by Speed Limiter text field when invalid value is
entered"), string];
        alternateButtonString = nil;
    }

    [proposedValueString = [NSString
localizedStringWithFormat:@"%f", proposedValue] retain];
    NSBeep();
    NSBeginAlertSheet(alertMessage, defaultButtonString,
alternateButtonString, otherButtonString, [self window],
self, @selector(speedSheetDidEnd:returnCode:contextInfo:),
NULL, proposedValueString, alertInformation);

    return NO;
}
```

The basic approach taken in this method is to test the value of the `error` parameter that was passed in from the delegate method, in order to determine whether you have a below-minimum, above-maximum or other illegal data entry. Based on this information, it then sets the string values of the error text and button name for the sheet, using local variables. Once the variables are set up, it is just a matter of passing them to the sheet using the same `NSBeginAlertSheet()` function defined in NSPanel that you used in *Recipe 3, Step 1*.

Now, examine the method in detail.

a. First, you declare the `alertInformation` local variable's string value once, since it will

be used without change no matter which error the user committed. The Aqua Human Interface Guidelines recommend more extensive communication with the user than was customary in the past. In addition to a descriptive error message in bold near the top of the alert, you should try to provide useful information in plain text in the middle of the alert to help the user understand what to do. You will provide the message text shortly to describe each of the three possible errors, but you provide informative text here to tell the user in every case that this text field requires a number between a minimum and a maximum value.

You use the now-familiar `NSLocalizedString()` function to define the key for the informative text and a comment that will be helpful to your localization contractors. You will provide the corresponding value for this key shortly in your `Localizable.strings` file. Your localization contractors will provide alternative versions of the string value for other languages.

Finally, you pass the `NSLocalizedString()` function's return value as a parameter to an invocation of the NSString `localizedStringWithFormat:` method. You use printf-style placeholders for the minimum and maximum values; these values will be filled in at run time by invoking the Speed Limiter slider's built-in `minValue` and `maxValue` methods. If your localization contractor uses Interface Builder to change the slider's minimum and maximum values, these error messages and informative strings will still function correctly, picking up the new limits automatically without requiring any change to your code. This is especially important for localization of a speed limiter control, since speed limits and units vary widely from country to country.

The `localizedStringWithFormat:` method is appropriate when placeholders represent values that need to be localized. Cocoa will automatically use the correct thousands and decimal separators, for example, depending on the preferred language set in the user's System Preferences. If your placeholders don't require localization, you can use the `stringWithFormat:` method, instead.

That may seem like it was a lot of work to set one variable, but in a single statement you have provided almost everything you need to make this string value fully localizable. This will become second nature to you in no time.

b. You next set the names of the default and other buttons, using the `NSLocalizedString()` function again to ensure that they can be localized. The Aqua Human Interface Guidelines recommend using names for dialog buttons that are descriptive of the action to be performed, when possible. Here, you use "Edit" instead of "OK" to return the user to the text field with the illegal value and the insertion point left in place. The value is not committed, but is left pending as if the user had not yet attempted to commit it. You use "Cancel" to cancel the invalid entry and restore the original entry, leaving it selected for immediate editing. These two buttons will be used in all branches of this method.

Note that the Cancel button is referred to as the "other" button and is passed as the fourth

parameter to the `NSBeginAlertSheet()` function. This "other" button appears immediately to the left of the default button, whether there are two or three buttons in the sheet. The "alternate" button, which you will define shortly, is the optional third button, which, in a three-button sheet, appears some distance to the left of the Cancel button. In a two-button sheet, you pass the alternate button as `nil` to suppress it.

It is easy to become confused about which is the "other" and which is the "alternate" button, since their names mean essentially the same thing. Indeed, O'Reilly's *Learning Cocoa* has them backwards in the example starting on page 142 of the first edition (May 2001). The O'Reilly example only works because it uses a two-button sheet, and the alternate button (Cancel, in their example) closes up alongside the OK button; if they had used a three-button example, the Cancel button would have appeared to the far left of the sheet. It doesn't help that, in the NSPanel header file, both the `NSAlertAlternateReturn` and `NSCancelButton` enumeration constants are equated to 0. Don't let this anomaly trouble you. In the context of the `NSBeginAlertSheet()` function, the `NSAlertAlternateReturn` constant is used to detect a user's having pressed the optional third button, and the `NSAlertOtherReturn` constant is used to detect the second button, which is usually Cancel. The `NSOKButton` and `NSCancelButton` constants are used for other purposes and their values are irrelevant here.

c. Now you turn to the three-branched test of the delegate method's error description. Here, you encounter the localization problem in reverse. The string returned by Cocoa in the `error` parameter to the `control:didFailToFormatString:errorDescription` delegate method is a meaningful description intended for possible display in the sheet itself; it is not a cryptic code useful only to programmers. If you are working on a machine configured for the English language, this delegate method will receive its `error` parameter in English. Your first problem is to figure out what these English string values might be, so that you can test them and branch according to the result; they aren't currently documented. Once you solve this problem (which you might do by simple experimentation), you have the even harder problem of finding out what values will be passed to you on machines configured for other languages. If you look at the NSControl documentation where the delegate method is described, you will learn only that the `error` parameter returns a "localized" string describing the error. For example, it might be in German if one of your users works in Germany.

   What to do if you want your application to be fully internationalized? If you've been studying Cocoa assiduously, you have begun to realize that just about everything relating to an application bundle or a framework bundle is located in the bundle itself—headers, documentation, localized language folders like `English.lproj` containing images and sounds, and what have you. It dawns on you that some of Cocoa's framework bundles might contain their own equivalents of your `Localizable.strings` file. Where would the strings for data formatting errors be located?—why, in the Foundation framework bundle, of course, where NSFormatter is defined.

Using the Finder, navigate to /System/Library/Frameworks/Foundation.framework/Resources/English.lproj. There, in plain sight if only you know where to look, is a file called `Formatter.strings`. Drag it to the Project Builder icon in the Dock to open it, and you see a perfectly ordinary set of key-value pairs for the error strings you are looking for. For example, the key "Maximum exceeded" is equated to the output string "Maximum exceeded". You will also see several other language folders in the bundle. Look in German.lproj, for example; you find that the key is still in English (this is typical of Cocoa), "Maximum exceeded", but the value is in German, "Maximum überschritten" (Latin is Latin in any language).

But do you really need to look up the string values manually for each language and code them verbatim into your tests? Of course not; Cocoa provides built-in methods to do this for you. All you need is the English-language value of the keys, which you do have to look up manually in the `Formatter.strings` file.

d. Looking at the code, above, you see that each branch tests for the value of the incoming error description using the standard NSString `isEqualToString:` method. However, to get the localized version of the description returned in the `error` parameter, you call upon the `NSLocalizedStringFromTableInBundle()` function declared in the NSBundle header file. In this function and its siblings, you identify the `.strings` file by passing it in by name (without the "`.strings`" extension) as `@"Formatter"`. You identify the bundle in which the `Formatter.strings` file lives by invoking NSBundle's `bundleForClass:` method, passing it NSFormatter's `class` method. In this way, you can be assured of obtaining the correct localized string value no matter where Apple may eventually place the Foundation framework bundle in Mac OS X's directory structure.

e. In each branch, you provide unique values for the message text and the third, or "alternate" button.

f. You dismiss the alert by returning `NO` as the method's return value. This rejects the attempted illegal entry—that is, it declines to commit it to the data variable in the MySettings model object—and, if you provide no new value, it leaves the insertion point active in the field awaiting the user's entry of the correct value.

The rest of the `sheetForSpeedTextFieldFormatFailure:errorDescription:` method uses the same techniques that are described above, until you get to the `NSBeginAlertSheet()` function. There, you will find the answer to your next question: How do you tell which button the user clicked to dismiss the sheet, and how do you do something with that information?

6. *Get the user's response to the sheet.* In earlier, prerelease versions of Mac OS X, a simple technique was used to get the user's response to a modal sheet: you called a function to present the sheet, and it returned a value representing the button that the user had clicked. However, this didn't work

properly when multiple sheets were open at once in different windows—a possibility that is implicit in the fact that a document-modal sheet is modal only with respect to the document, not other documents or the rest of the application. Cocoa got confused and sometimes dismissed the wrong sheet.

In the last developer preview version of Mac OS X, a new technique was introduced, and in the Public Beta release, the relevant methods were enhanced. Now, the new `NSBeginAlertSheet()` function returns control to the application immediately upon presenting the sheet, without waiting for the user to press a button; your application's flow of execution continues while the sheet just sits there, frozen, with a document-modal sheet in front of it. You must declare and specify a callback method to obtain the information you need regarding the user's actions in the sheet and to bring it back to life. Cocoa will invoke the callback method for you when the user finally gets around to pressing a button in the sheet. Between the time the `NSBeginAlertSheet()` function is called and the time the callback method is called, the document window remains in a state of suspended animation while the rest of the application works around it. This new technique may seem very complex, but you will get used to it. The code is summarized in the next paragraph, then explained in the following paragraphs. (Note that the NSPanel class reference document and the Foundation function reference document have not yet, at this writing, caught up with the code. You must examine the NSPanel header file itself to see the current declaration of the `NSBeginAlertSheet()` function; the Foundation function reference has the names of the selector parameters wrong.)

You designate one of your custom application classes as the temporary "modal delegate" for this sheet. The modal delegate may optionally implement one or both of two callback routines, the format for which is specified in NSPanel. Each of these callback methods will know the identity of the button the user clicked and is responsible for acting on that information. You specify the modal delegate class in the `modalDelegate` parameter to the `NSBeginAlertSheet()` function, and you also pass references to either or both of your two callback selectors, in the `didEndSelector` parameter and the `didDismissSelector` parameter, respectively. Each of these two parameters, if not `NULL`, must refer to a callback method implemented in your designated modal delegate class using a special signature described in the NSPanel header file, namely, `sheetDidEnd:returncode:contextInfo:` and `sheetDidDismiss:returncode:contextInfo:`. These methods can be named anything you like, as long as they follow the prescribed signature.

Here, you designated MyWindowController as the modal delegate, by passing `self` as the `modalDelegate` parameter to the `NSBeginAlertSheet()` function. You also invoked a custom `speedSheetDidEnd:returncode:contextInfo:` method in that function, as the selector for the `didEndSelector` parameter (passing `NULL` in the `didDismissSelector` parameter). Now you must implement that method.

In `MyWindowController.h`, declare the method as follows, at the end of the Formatter errors section:

```
- (void)speedSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo;
```

In `MyWindowController.m`, define the method as follows:

```
- (void)speedSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo {
    if (returnCode == NSAlertOtherReturn) {
        [[self speedTextField] abortEditing];
        [[self speedTextField] selectText:self];
    } else if (returnCode == NSAlertAlternateReturn) {
        [self updateTextField:[self speedTextField]
setting:(NSString *)contextInfo];
        [self speedTextFieldAction:[self
speedTextField]];
    }
    [(NSString *)contextInfo release];
}
```

Here's what's going on. The `NSBeginAlertSheet()` function, in passing a reference to the `speedSheetDidEnd:returnCode:contextInfo:` selector in the `didEndSelector` parameter, also populated that method's first two parameters with values you might need, namely, a reference to the sheet and the return code of the button the user clicked. It also populated the third parameter of the method, `contextInfo`, with any context information you passed in the `contextInfo` parameter of the `NSBeginAlertSheet()` function. The `speedSheetDidEnd:returnCode:contextInfo:` method uses some of this information to recognize which button the user clicked and to reset the value of the text field accordingly. The method is called only when the user clicks one of the buttons in the sheet. This may occur minutes, hours, or days after the sheet was presented, since this sheet is modal only to this document window. The user may continue to work in other windows or even in other applications while the sheet is still pending.

a. In the `speedSheetDidEnd:returnCode:contextInfo:` method, you haven't used the selector's reference to the sheet. You could use it to dismiss the sheet yourself in appropriate circumstances using NSWindow's `orderOut:` method (for example, if you want to dismiss both the sheet and the underlying window simultaneously, as the save and open sheets do), but here you'll just rely on the `NSBeginAlertSheet()` callback mechanism to dismiss the sheet.

b. You use the `returnCode` parameter to test which button the user clicked, using the

constants declared in NSPanel for this purpose, `NSAlertDefaultReturn`, `NSAlertAlternateReturn`, and `NSAlertOtherReturn`. (As noted above, you should not use the `NSOKButton` and `NSCancelButton` constants here, because they are declared for a different purpose. Indeed, these two constants have values that conflict with those used by the `NSBeginAlertSheet()` method.)

You don't care if the user clicked the "default" button ("Edit"). By doing nothing in that case, you simply rely on the `NSBeginAlertSheet()` callback mechanism to reject the invalid data entry and leave the insertion point active for editing in the text field. To understand how this works, you need to understand the subtlety involved in the statement, at the beginning of this instruction 6, to the effect that the `NSBeginAlertSheet()` function "returns control" immediately. This means only that the application's flow of control immediately goes on to other tasks. It does not mean that the `sheetForSpeedTextFieldFormatFailure:errorDescription:` or `control:didFailToFormatString:errorDescription` methods return their results immediately. To the contrary, Cocoa has holds not only the text field and the sheet, but also these methods, in suspension while your sheet is pending, waiting to see the result of the callback method. Once the user finally clicks a button in the sheet, the `sheetForSpeedTextFieldFormatFailure:errorDescription:` method passes its `NO` return value to the `control:didFailToFormatString:errorDescription:` method, which also returns `NO`, thereby rejecting the bad string.

If the user clicked the "other" button ("Cancel"), the method invokes NSControl's `abortEditing` method. This restores the text field's displayed value to its original value before editing of the field began, which was the current, unchanged data value in MySettings. Notice that the method does not have to access the MySettings object to obtain the original value; that value is remembered by the window's field editor during this editing session. The field editor is a shared object in the window that is normally used to hold and manipulate text in the currently active text object in the window, often a text field. The method also invokes NSTextField's `selectText:` method to select the restored text, leaving it in a state for immediate revision by the user.

Finally, if the user clicked the "alternate" button ("Set to 75 mph" or "Set to 155 mph"), the method invokes MyWindowController's custom `updateTextField:setting:` method, which you wrote long ago, to force the field to display the minimum or maximum value, then calls the text field's action method, which you also wrote earlier, to set the data value in MySettings to match. The theory of these alternate buttons is that the user's entering an out-of-range value may have been an attempt to enter the slider's minimum or maximum value. The action method registers this change with the Undo Manager, just as it would if it had been invoked by the user's entering the minimum or maximum value directly into the text field in the first place.

c. The `contextInfo` parameter also needs explanation. Normally, you should consider

using a temporary dictionary object to pass context information through the `contextInfo` parameter, to make it easier to revise the application in the future if you discover a need to pass more or different information. Here, however, for convenience, you have passed a single value, an NSString object representing the minimum or maximum allowed value, depending on whether the user typed a value in the text field that was below its minimum or above its maximum.

It is important to notice that, in creating the NSString object in the `sheetForSpeedTextFieldFormatFailure:errorDescription:` method, you retained it, and in the `speedSheetDidEnd:returnCode:contextInfo:` method you released it. You had to retain it, because otherwise it would have been released automatically by Cocoa when you exited from the `sheetForSpeedTextField:errorDescription:` method. Objects returned by convenience methods like NSString's `localizedStringWithFormat:` method are always autoreleased before they are returned to you, and they are therefore always released when the autorelease pool is released, unless you explicitly retain them. It's worth compiling and running the application without the `retain` just to see what happens. As soon as execution enters the `speedSheetDidEnd:returnCode:contextInfo:` method and attempts to obtain the value of the `contextInfo` parameter, your program crashes because the value is no longer guaranteed to be accessible in memory. By the same token, you must balance the `retain` with a `release` when you are done with the value in the `speedSheetDidEnd:returnCode:contextInfo:` method, in order to avoid a memory leak.

A final detail: When the `proposedValueString` variable was passed in the `contextInfo` parameter, it was cast from an NSString object to a void pointer (`void*`). When it was received in the `speedSheetDidEnd:returnCode:contextInfo:` method, you had to cast it back to an NSString value in order to be able to release it at the end of that method. Objective-C will not act on void pointers.

7. Next, add all of the required entries to your `Localizable.strings` file. Basically, just include an entry for every unique invocation of the `NSLocalizableString()` function. You should not include an entry for the `NSLocalizedStringFromTableInBundle()` function invocations, since those entries already exist in Cocoa's `Formatter.strings` file. In this case, the new entries are the following, placed in the Alert strings section:

```
/* Common buttons */
/* Name of OK button */
"OK" = "OK"
/* Name of Cancel button */
"Cancel" = "Cancel"

/* Speed Limiter text field alert */
/* Message text for alert posed by Speed Limiter text
field when value smaller than minimum is entered */
"%@ mph is too slow for the Speed Limiter." = "%@ mph
is too slow for the Speed Limiter."
/* Message text for alert posed by Speed Limiter text
field when value larger than maximum is entered */
"%@ mph is too fast for the Speed Limiter." = "%@ mph
is too fast for the Speed Limiter."
/* Message text for alert posed by Speed Limiter text
field when invalid value is entered */
"“%@” is not a valid entry for the Speed Limiter." =
"“%@” is not a valid entry for the Speed Limiter."
/* Informative text for alert posed by Speed Limiter
text field when invalid value is entered */
"The Speed Limiter must be set to a speed between %1.1f
mph and %1.1f mph." = "The Speed Limiter must be set to
a speed between %1.1f mph and %1.1f mph."
/* Name of alternate button for alert posed by Speed
Limiter text field when value smaller than minimum is
entered */
"Set %1.1f mph" = "Set %1.1f mph"
/* Name of alternate button for alert posed by Speed
Limiter text field when value larger than maximum is
entered */
"Set %1.1f mph" = "Set %1.1f mph"
```

8. Finally, in Interface Builder, Control-drag from the Speed Limiter text field to the File's Owner icon. In the NSTextField Info palette, select delegate in the left pane of the Outlets section, then click the Connect button. If you leave out this essential step through forgetfulness (a not uncommon experience!), none of this will work as expected and it may take you a while to figure out why.

Compile and run the application to test the sheet. With the Sliders tab view item selected, enter various legal and illegal values into the Speed Limiter text field. If you enter a legal value, the text field and the slider will reflect it, as before. If you enter an illegal value less than the minimum value in the acceptable range, a beep will sound and a sheet will descend explaining exactly what is wrong and, in accordance with the new human interface guidelines for Aqua, giving you useful information about how to solve the

problem. Notice that, if you click Edit, you are returned to an active insertion point in the text field, waiting for you to revise the entry. If you click Cancel, the previous value is reinstated and selected in case you want to attempt to edit it again. In both cases, the slider remains set to its previous value, as it should, because no new value has been committed.

There is a third button, offering to let you set the value to the legal minimum. If you had entered an illegal value greater than the maximum, this button would offer to let you set the value to the legal maximum, instead. In either case, clicking this alternate button will enter the legal minimum or maximum value into the text field, as indicated, and the slider will snap to that new value because it has been committed.

Be sure to try entering a string, such as **`fast`**, into the text field. A somewhat different sheet descends, explaining that a number is required and offering only two buttons, Edit and Cancel.

Notice that Cocoa now, in Mac OS X 10.0, disables the close button on a window while a sheet is pending, and it disables several menu items, as well, which otherwise could be used to do things to the window in violation of the document modality principle of sheets.

After trying several illegal values and fixing them, exercise the Undo and Redo commands repeatedly to make sure they work as expected.

Open several windows at once and attempt to enter illegal values in each of them in turn, pressing the Enter key—but don't fix any of them yet; that is, leave all of the sheets open at once. Confirm that, although a document-modal sheet remains open in each window preventing user interaction in that window, user interaction is nevertheless allowed in other windows that do not have a pending sheet, as well as in other applications. Also confirm that the correct sheet is dismissed when you resolve the illegal entry in each window and dismiss its sheet, in a random order.

Finally, try editing the text field in several windows at once *without* pressing the Enter key. That is, leave edits pending in several windows at the same time. Everything will work properly. You can enter a value in one window's text field by hitting the Enter key or using its slider, and the text fields in all the other windows will remain pending. If you now close any of the windows with pending entries, or quit the application, the pending entries will be discarded because the user has not yet affirmatively commmitted them.

Unfortunately, however, you haven't yet tried all the outlandish things that you should assume your users will try, so you aren't out of the woods. For example, harkening back to *Recipe 3, Step 2*, try deleting whatever is in the text field, then hitting the Enter key while the field is blank. Uh, oh! A value of "0.0…" has appeared in the text field, and, worse yet, it has been committed without any warning that it is out of range. Then try typing **`0.0`** into the text field and hitting Enter. Again, an illegal value has been accepted and committed (although, oddly, typing **`0`** is not accepted). For variety, try switching to another tab view item using the navigation buttons that you created earlier while an illegal entry, such as **`45`**, is pending. A sheet is presented signalling the attempted illegal entry, but while the sheet is pending, the window underneath switches to the new tab view item, in violation of human interface guidelines specifying that a

document-modal sheet must not permit user actions in the window. Except in the case of a blank or 0.0 entry, Cocoa prevents you from changing tab view items by clicking on a tab while an illegal value is pending, but it does so in a way that fails to offer the same benefit to your navigation buttons.

In the next few Steps, you will add code to the application to resolve these remaining problems.

Introduction > Contents > Recipe 4 > Step 2 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 4: User controls—Text fields (sheets)

Step 3: A generic document-modal sheet to prevent deletion from a text field

- Highlights:
  - Intercepting an attempt to commit a change to a text field

In many cases, if not most, it is desirable to allow a user to delete information found in a text field. Consider the typical form, spreadsheet or database. If a user's phone is disconnected or other data becomes inapplicable, the user must be able to erase the data. Selecting the contents of a text field, hitting the Delete key, and pressing Enter is an everyday sequence of actions while sitting at a computer.

When a text field is designed to contain free-form text, implementing the ability to delete its contents is easy: just do nothing. No formatter need be attached to a free-form field, and a blank entry—that is, an empty or zero-length string—is as acceptable as any other text entry. Often, however, a text field is formatted according to an attached formatter, yet deleting its contents to leave the field blank is still desirable. For example, most web forms allow deletion of information even from required fields, deferring the validity test to the time when the Submit button is clicked and the entire form can be checked at once. It is presumably for this reason that, as you have discovered, an attached formatter is not invoked by Cocoa

when data is deleted from a field, but only when data is inserted. Thus, even for fields with an attached formatter, allowing the user to delete the fields' contents requires no code.

The fact that a formatter is not invoked when deleting information creates an issue, however, on those occasions when it is necessary to prevent a user from deleting the contents of a text field. In the case of the Speed Limiter text field, for example, the slider is constrained to a range of values from 75 mph to 155 mph, and the text field is expected to match the slider's displayed value at all times. The field is initialized to the default value of the slider when the window opens, and a user should not be permitted to leave the field blank under any circumstances. As you have discovered, you can't rely on the attached formatter to detect and prevent a blank entry, since a formatter is not invoked in that case.

The issue is even more complex in the case of a text field that is formatted as a floating point number, like the Speed Limiter field. You may wonder how it happens that entering a blank value in the Speed Limiter field results in a display of "0.0…." The answer has to do with the fact that, somewhere in the process of updating the text field on screen to reflect the blank entry, NSString's `floatValue` method is called (your `speedTextFieldAction:` method explicitly invokes the text field control's `floatValue` method through the `sender` parameter, and the NSControl reference document indicates that this is converted to a string, presumably through a call to NSString's `floatValue` method, via NSControl's `validateEditing` method). One aspect of NSString's `floatValue` behavior, according to the NSString class reference document, is that it returns 0.0 if the string doesn't begin with a valid text representation of a floating point number. Since an empty string doesn't begin with anything, Cocoa interprets a blank entry as 0.0. This is common practice among programming languages, because it allows a default "empty" value to be stored in a variable that is typed as a float. This documented but perhaps obscure side effect of the `floatValue` method has to be taken into account when allowing deletion of the contents of a field that is formatted as a float. In this case, the value will be kept as zero in the MySettings instance variable, `speedValue,` and displayed as "0.0…." Because the formatter is not invoked, the value is displayed in the Speed Limiter text field with many trailing zeros to the right of the decimal point, an undesirable result for fields whose formatter specifies, as this one does, only one trailing zero.

The formatter you have attached to the Speed Limiter text field simply isn't being invoked when the user deletes the field's contents. You can confirm this by setting a breakpoint on the `control:didFailToFormatString:errorDescription:` method in `MyWindowController.m` and running the application in debug mode, then attempting to enter a blank value; the breakpoint is never hit, and "0.0…" appears in the field. Oddly, the same is true if you explicitly enter **`0.0`**, although why the formatter is not invoked in this case is not clear. It would seem to be a bug in Cocoa; after all, the formatter is invoked if you enter **`0`**.

A well-designed application should not allow users to employ such an ambiguous means of committing a zero value as simply making a blank entry, without disclosing this result. So displaying "0.0" when a floating point text field is emptied seems reasonable, although one might prefer to store and display the field as a string when allowing deletion in such a field. The problem of formatting a zero value in a floating point text field is deferred for now, since you will not allow zero values here; perhaps you will take up the

problem at a later time.

It is clearly wrong to accept a blank entry, or to accept an explicit 0.0 entry, here, in the face of the range restriction that applies to the Speed Limiter text field. In this Step, therefore, you will write some generic routines to suppress blank entries—that is, to prevent deletion of data in a text field—and at the same time you will write some generic routines to deal with the more specialized problem of an explicit out-of-range 0.0 entry. You will create separate but very similar routines to handle both situations, presenting sheets in each case to inform the user of the problem. The routines will be written generically, in order to allow their reuse by any other text fields that might have similar constraints.

Note that it might be less disruptive to your users' work patterns to dispense with a sheet in these situations, and simply to restore the text field to its previous value silently. That is how TextEdit behaves, for example, when an attempt is made to delete the values in the text fields for window width and height in its Preferences dialog. Here, it is perhaps even more obvious that zero is not an acceptable value, since the Speed Limiter is associated visually and interactively with a slider that has a prominently-displayed minimum acceptable value of 55 mph. However, other situations might not be so clear, and this is a Recipe about creating interactive sheets, so that's what you'll do here.

1. Since NSNumberFormatter isn't being invoked, you must hunt around for some other hook to catch attempts to enter a blank text value or 0.0. In NSControl, you find just the thing, the `control:textShouldEndEditing:` delegate method. If this delegate method is implemented in the control's delegate, it will be invoked whenever the user attempts to do something that would cause the text field to relinquish first responder status, such as hitting the Return, Enter or Tab key—unless an attached formatter intercepts the flow of control first. The `control:textShouldEndEditing:` method is typical of the many delegate methods provided for you in the Cocoa frameworks. You see here how important it is to read the documentation and become familiar with them, because they often make it easy to accomplish important tasks.

   You have already designated MyWindowController as the Speed Limiter text field's delegate, so you will implement this delegate method in MyWindowController. Because it can be triggered by any text field that similarly appoints MyWindowController as delegate, its first task is to identify the responsible text field. In `MyWindowController.m`, define the delegate method as shown below, at the top of the Input validation and formatting methods section. You do not need to declare this method in the header file because it is a delegate method.

```
// Methods to force completion and validation of pending
edits

- (BOOL)control:(NSControl *)control
textShouldEndEditing:(NSText *)fieldEditor {
    if (control == [self speedTextField]) {
        if ([[fieldEditor string] length] == 0) {
            return [self sheetForBlankTextField:control
name:NSLocalizedString(@"Speed Limiter", @"Name of Speed
Limiter text field")];
        } else if ([[fieldEditor string] floatValue] ==
0.0) {
            return [self sheetForZeroTextField:control
name:NSLocalizedString(@"Speed Limiter", @"Name of Speed
Limiter text field")];
        } else {
            return YES;
        }
    } else {
        return YES;
    }
}
```

A Cocoa delegate method with the word "should" in its name is usually a request for permission to perform some work. If you implement the delegate method and return `NO`, you have exercized your veto power and the work will not be performed. You must return `YES` to allow the work to go forward. Here, each of the `sheet…` methods, which you will write in a moment, will return `NO` to prevent entry of the invalid value, but they will apply different modifications to the contents of the text field depending on the user's response to the sheet. However, if the value the user entered is valid (that is, it was not intercepted by the formatter because it is within the prescribed range), then this method will return `YES` to allow the entry to be committed. It will also return `YES` if some other field is key, because other fields may appropriately allow deletion of their contents.

After determining that it is the Speed Limiter text field which is about to end an editing session, this method performs two tests to determine which sheet to present. The first test equates the length of the string to 0. Remember that Cocoa relies heavily on Unicode, where individual characters can be represented by multi-byte sequences. Testing whether the length is 0, however, is a safe way to be sure the field is blank. The second test is used here only to catch a field in which the user has explicitly entered 0.0. If Cocoa's employment of formatters should eventually be revised to catch this value when it is out of range, this second test and the `sheetForZeroTextField:name:` method could be eliminated.

2. Next, you must implement the custom `sheetForBlankTextField:name:` and

`sheetForZeroTextField:name:` methods you just invoked, each presenting a sheet using the same techniques you used in *Step 2*. You will design this method to be reused by any and all text fields for which a value is required or a 0.0 entry is to be suppressed. The alert messages and informative text are therefore very general, but you provide for an option to pass a name in the `name` parameter to customize the sheet to some extent. You can pass `NULL` if the field has no name or you don't wish to show it in the sheet.

In `MyWindowController.h`, declare the methods as follows, at the top of the Input validation and formatting methods section:

```
// Methods to force completion and validation of pending
edits

- (BOOL)sheetForBlankTextField:(NSControl *)control
name:(NSString *)fieldName;

- (BOOL)sheetForZeroTextField:(NSControl *)control
name:(NSString *)fieldName;
```

Define them in `MyWindowController.m`, after the `control:textShouldEndEditing:` method in the Methods to force completion and validation of pending edits section:

```
- (BOOL)sheetForBlankTextField:(NSControl *)control
name:(NSString *)fieldName {

    NSString *alertMessage;

    NSString *alertInformation = NSLocalizedString(@"A value
must be entered in this field.", @"Informative text for alert
posed by any text field if empty when attempting to resign
first responder status");
    NSString *defaultButtonString =
NSLocalizedString(@"Edit", @"Name of Edit button");
    NSString *otherButtonString =
NSLocalizedString(@"Cancel", @"Name of Cancel button");

    if (fieldName == NULL) {
        alertMessage = NSLocalizedString(@"The field is
blank.", @"Message text for alert posed by any unnamed field
if blank when attempting to resign first responder status");
    } else {
        alertMessage = [NSString
stringWithFormat:NSLocalizedString(@"The %@ field is blank.",
```

```
@"Message text for alert posed by any named text field if
blank when attempting to resign first responder status"),
fieldName];
    }

    NSBeginAlertSheet(alertMessage, defaultButtonString, nil,
otherButtonString, [self window], self,
@selector(blankTextFieldSheetDidEnd:returnCode:contextInfo:),
NULL, control, alertInformation);

    return NO;
}

- (BOOL)sheetForZeroTextField:(NSControl *)control
name:(NSString *)fieldName {

    NSString *alertMessage;

    NSString *alertInformation = NSLocalizedString(@"A value
of 0.0 is not valid in this field.", @"Informative text for
alert posed by any text field if 0.0 when attempting to
resign first responder status");
    NSString *defaultButtonString =
NSLocalizedString(@"Edit", @"Name of Edit button");
    NSString *otherButtonString =
NSLocalizedString(@"Cancel", @"Name of Cancel button");

    if (fieldName == NULL) {
        alertMessage = NSLocalizedString(@"The field value is
0.0.", @"Message text for alert posed by any unnamed field if
0.0 when attempting to resign first responder status");
    } else {
        alertMessage = [NSString
stringWithFormat:NSLocalizedString(@"The %@ field value is
0.0.", @"Message text for alert posed by any named text field
if 0.0 when attempting to resign first responder status"),
fieldName];
    }

    NSBeginAlertSheet(alertMessage, defaultButtonString, nil,
otherButtonString, [self window], self,
@selector(zeroTextFieldSheetDidEnd:returnCode:contextInfo:),
NULL, control, alertInformation);

    return NO;
```

```
}
```

As in *Step 2*, the `NSBeginAlertSheet()` function here designates MyWindowController as modal delegate and passes in a `didEndSelector` reference so that the user's choice of buttons in the sheet can be processed. The alternate button is passed as `nil` because this is a two-button sheet. Instead of passing a string object in the `contextInfo` parameter, it passes a reference to the text field object. Since the text field object is already instantiated, there is no need to bother with `retain` and `release`.

3. Now declare the `didEndSelector` methods in `MyWindowController.h` after the `sheet`… methods, as follows:

```
- (void)blankTextFieldSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo;
- (void)zeroTextFieldSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo;
```

Define the methods in `MyWindowController.m` after the `sheet`… methods, as follows:

```
- (void)blankTextFieldSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo {
    if (returnCode == NSAlertOtherReturn) {
        NSTextField *field = (NSTextField *)contextInfo;
        [field abortEditing];
        [field selectText:self];
    }
}

- (void)zeroTextFieldSheetDidEnd:(NSWindow *)sheet
returnCode:(int)returnCode contextInfo:(void
*)contextInfo {
     if (returnCode == NSAlertOtherReturn) {
        NSTextField *field = (NSTextField *)contextInfo;
        [field abortEditing];
        [field selectText:self];
    }
}
```

Notice that `contextInfo` is cast from a void pointer to an NSTextField object here and assigned

to a local variable in order to invoke the appropriate methods.

4. Don't forget to add the localized strings used in these methods to the `Localizable.strings` file.

```
/* Name of Edit button */
"Edit" = "Edit"

/* Field names */
/* Name of Speed Limiter text field */
"Speed Limiter" = "Speed Limiter"

/* Blank text field alert */
/* Message text for alert posed by any unnamed field if
blank when attempting to resign first responder status
*/
"The field is blank." = "The field is blank."
/* Message text for alert posed by any named text field
if blank when attempting to resign first responder
status */
"The %@ field is blank." = "The %@ field is blank."
/* Informative text for alert posed by any text field if
blank when attempting to resign first responder status
*/
"A value must be entered in this field." = "A value must
be entered in this field."

/* Zero text field alert */
/* Message text for alert posed by any unnamed field if
0.0 when attempting to resign first responder status */
"The field value is 0.0." = "The field value is 0.0."
/* Message text for alert posed by any named text field
if 0.0 when attempting to resign first responder status
*/
"The %@ field value is 0.0." = "The %@ field value is
0.0."
/* Informative text for alert posed by any text field if
0.0 when attempting to resign first responder status */
"A value of 0.0 is not valid in this field." = "A value
of 0.0 is not valid in this field."
```

Compile and run the application to test the sheets. With the Sliders tab view item selected, try deleting whatever is in the text field, then hitting the Enter key. Confirm that a sheet warns you that you must

provide a value. Also try entering 0.0 in the field.

Introduction > Contents > Recipe 4 > Step 3 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 4: User controls—Text fields (sheets)

Step 4: Preventing tab view navigation while an illegal text entry is pending

- Highlights:
  - Human interface considerations relating to invalid text field entries

At this point, you have in place good validity checking for the text field when the user hits the Return, Enter, or Tab key, and it would also work properly if another text field were added to the Sliders pane and the user clicked into it. But there is still another situation, alluded to previously, in which the text field's behavior should be reviewed.

The issue arises when the user attempts to switch to another tab view item. Cocoa's default behavior is to treat the attempted selection of a second tab view item as a signal to validate and commit any pending edit in the first tab view item; that is, to attempt to force the text field in which the edit is pending to resign first responder status. You can see the effect of this by clicking another tab while a blank or out-of-range value is pending in the text field: the sheets you created previously are immediately presented, as if you had pressed the Enter key or clicked into another text field. You can also see this in action by setting a breakpoint on the `control:textShouldEndEditing:` and `control:didFailToFormatString:errorDescription:` delegate methods. When you run the application in the debugger while a blank or out-of-range value is pending in the text field, you see that either of these methods is invoked when you click on another tab. In other words, a text field is tested for validity whenever the user attempts to switch tab view items, and the switch is prevented if the field is invalid.

Surprisingly, however, if you attempt to switch tab view items using the navigation buttons you added in *Recipe 2, Step 8*, the destination tab view item appears even if the pending text field entry in the first tab view item is blank or otherwise illegal. Your users see your sheet telling them about the illegal entry in the text field, but the tab view item is nevertheless switched out from under them while the sheet is open and before they can respond to it. The navigation button's action method is apparently invoked before Cocoa can present the sheet and place the document into its modal state. This is a violation of the human interface guidelines, which

describe document-modal sheets as blocking all user actions on a window until they are dismissed. Cocoa in general takes care of this for you, for example, by disabling the close box on a window and disabling several menu items while a sheet is pending. You must now do something to make the navigation buttons behave identically to the built-in behavior of the tabs.

One solution would be to add code to the navigation button action methods to force any text field having keyboard focus to resign first responder status, and to allow the selection of another tab view item to proceed only if this is successful. A standard way to force validation of a pending text field entry in these circumstances is to call NSWindow's `makeFirstResponder:` method to designate the window itself as first responder, implicitly forcing the text field to resign first responder status. This method returns `YES` if successful, so it would make a perfect test in the navigation button action methods. However, there is one, perhaps minor, downside to doing it this way. If you later add additional user controls to switch tab view items, such as the pull-down menu button or radio button cluster recommended as options in the human interface guidelines, you would have to remember to implement this same test in each of their action methods, as well. It would be nice to avoid such redundant code by finding a single approach to cover all cases.

If you are beginning to get the drift of Cocoa delegate methods, you know you should look for a delegate method in NSTabView that will let you intercept and thwart the attempt to select another tab view item if an illegal entry is pending. You don't need a means to disable the navigation buttons while the sheet is pending, because Cocoa implements the "modal" in "document-modal sheet" for you: all controls in the window become unresponsive once a sheet is pending on that window. Cocoa does not put the controls into a visually disabled state, presumably because the sheet is so obvious (Cocoa does disable some menu items that could otherwise be used to affect the window while the sheet is pending, presumably because the association between the window and the menu items is not so obvious).

The delegate method you need to force a validity check on the field is NSTabView's `tabView:shouldSelectTabViewItem:` method. This is similar to the `tabView:willSelectTabViewItem:` method that you invoked in *Recipe 2, Step 7* to enable and disable the navigation buttons selectively in the event the user was selecting the first or last tab in the tab view. Here, however, you use the "should" variant of the delegate method because you want to be able to veto the switch.

In `MyWindowController.m`, define the delegate method as shown below, at the top of the Methods to force completion and validation of pending edits section. As a delegate method, it does not need a declaration.

```
- (BOOL)tabView:(NSTabView *)tabView
shouldSelectTabViewItem:(NSTabViewItem *)tabViewItem {
    return [[self window] makeFirstResponder:[self window]];
}
```

Invoking the `makeFirstResponder:` method on the window here is exactly equivalent to invoking it in each of the navigation button action methods, which are invoked by your navigation buttons and will be invoked by any other tab-switching user controls or menu items you may later add to the application. The attempt to force the window itself to become the first responder immediately triggers either the `control:textShouldEndEditing:` delegate method in the case of a blank entry or the `control:didFailToFormatString:errorDescription:` delegate method in the case of an illegal

entry, because it implicitly tells the text field to resign first responder status. Since either of these methods will return `NO` in the case of a blank or illegal entry, preventing the text field from relinquishing first responder status, the command to make the window first responder will fail and `tabView:shouldSelectTabViewItem:` will also return `NO`. The end result is to overrule the attempted selection of a different tab view item. The sheet that was posed lets the user decide how to resolve the issue, whereupon the user can again switch tab view items.

Human Interface Considerations

What becomes of data that is pending in a text field, if the user selects another tab view item, or closes the window or quits the application, before pressing the Enter key to commit it?

Cocoa's default behavior is to perform a validity check on a pending edit when a new tab view item is selected. This makes it clear that switching panes in a tab view is regarded as a signal that the user intends to commit any pending edit. The user is not interrupted with a notice that the data will be committed, because this is the way things are supposed to work.

This seems sensible from a human interface perspective. Switching panes does not have the same negative connotation as closing the window or quitting the application; it does not suggest that the user is abandoning work on the document and wants to discard pending edits, as those other two actions do. At the same time, it does suggest an intent to move on to other aspects of the document.

The alternative, leaving an edit session "live" in a text field that is hidden from view by another tab view item, could become confusing. After a while, the live text field would tend to be forgotten, yet at any moment the user's clicking in a text field in the current tab view item would force the other text field to commit, since only one text field can have focus in a window. Better to get it out of the way as soon as the user switches to a new pane. The user is protected in several ways: the user is still within the same window and can therefore easily return to the first tab view item to check the value; the Undo menu item allows the user to restore the previous value of the text field, even without returning to its tab view item; and the user will be asked by built-in Cocoa routines whether to save changes when closing the window.

It turns out, however, that the version of the Aqua Human Interface Guidelines that is current as this is being written advocates this latter, alternative behavior for dialogs, saying "In a dialog that has multiple panes (selected by tabs or a pop-up menu), avoid validating data when a user switches from one pane to another." No explanation is given, and nothing is said about ordinary application windows. Given that Cocoa's default behavior is to validate a pending text field when another tab is selected, you will, at least for now, accept the Vermont Recipes application's behavior as proper behavior for an application window.

A similar user interface issue arises when the user attempts to close the window holding the text field, or to quit the application. Closing a window or quitting after the user commits a change to a text field of course causes Cocoa to ask the user whether to save it. However, a pending, or uncommitted edit to a text field is simply discarded by default when the window is closed or the application quits.

Cocoa allows an attempt to close the window or to quit while an edit is pending, legal or illegal, without giving

any warning that the pending edit will be discarded and without offering to let the user save it. The unfinished edit in the text field is aborted and the new value is lost. The theory implicit in this behavior is that data isn't data until the user presses Enter or takes other steps recognized as evidence of an intent to end editing and commit the value. As noted above, switching tab view items is regarded as evidence of such an intent, but closing a window or quitting a document is regarded as evidence of an intent to abandon pending edits. This behavior is deeply ingrained in Cocoa. A few examples: the Window menu places a check mark in front of unsaved window names only after a change to a text field is committed; the automatic save sheet descends only when closing a window in which changes are committed; the review-all-changes dialog is posed when quitting only if changes have been committed; the close button in a window's title bar only acquires its dot when changes have been committed; and the Undo menu item is enabled only when changes have been committed. In the context of these interface signals, a pending, uncommitted edit simply isn't recognized.

If you were to attempt to override Cocoa's default behavior when a window is closed or the user quits, you would discover that there is no straightforward way to do so. There are methods to intercept these events. Examples are NSWindow's `windowShouldClose:` delegate method, NSDocument's `shouldCloseWindowController:delegate:shouldCloseSelector:contextInfo:` method, and NSApplication's `applicationShouldTerminate:` delegate method. You can use all of them to catch invalid pending edits before the window closes or the application quits, but doing so makes little sense if you can't also commit a valid pending edit and give the user the option to save it. Cocoa apparently invokes all of these methods too late in the process of closing a window or quitting the application to raise a save sheet or dialog on a pending, valid edit that wasn't committed until one of these methods was invoked.

So, enough is enough; you should do it Cocoa's way.

Try switching to another tab view item while an illegal entry is pending, to confirm that you are prevented from doing so until the entry is fixed. Try closing the window or quitting while an illegal entry is pending, and you will see that Cocoa instantly obeys your command.

You have now learned enough about sheets to make use of them throughout your application. Before moving on to additional text field examples in *Recipe 6*, however, you will first take another detour to study text field formatting in the next Recipe, *Recipe 5*.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 5: User controls—Text fields (formatters)

Download the project files for Recipe 5 as a disk image and install them

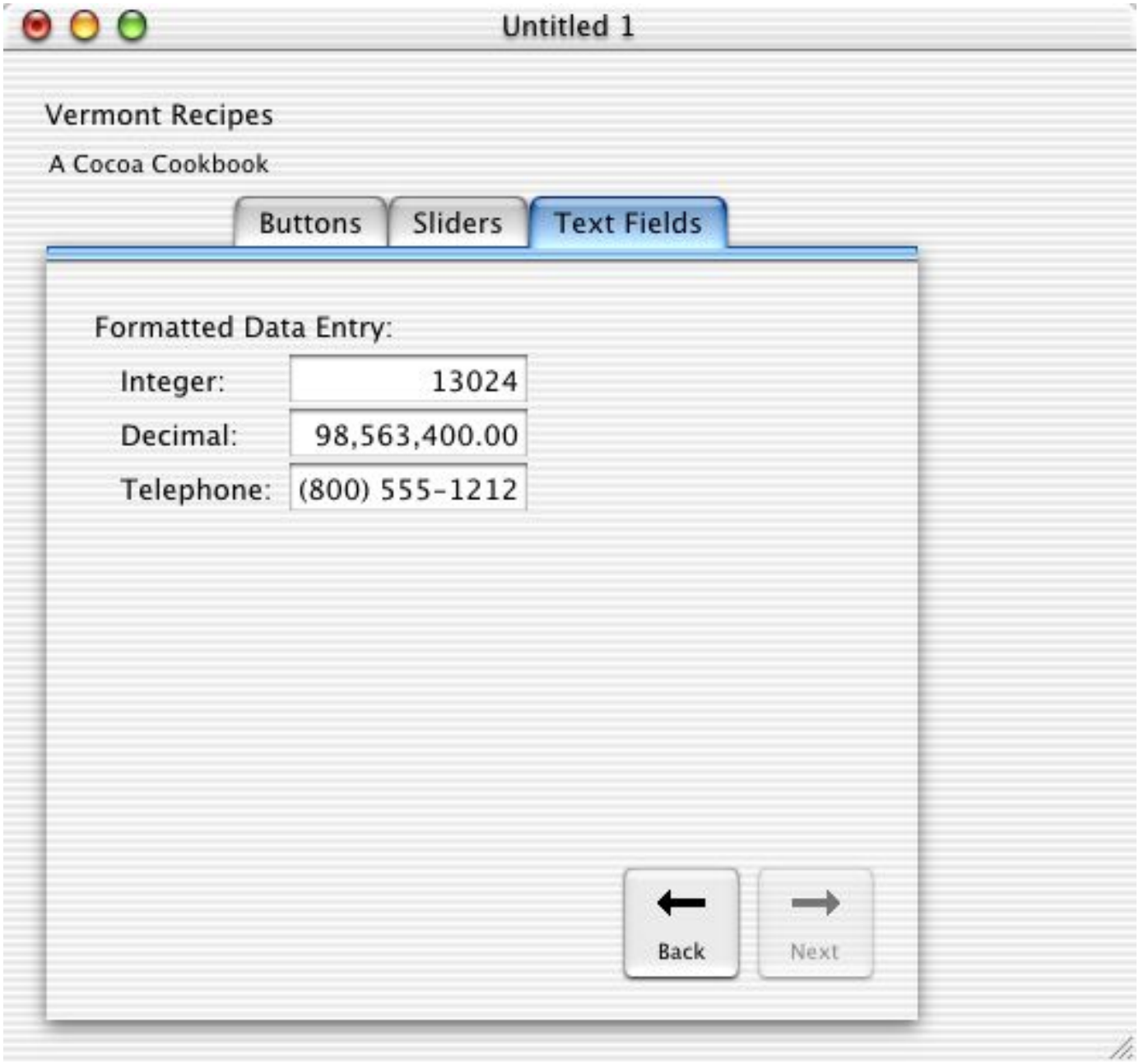
Download a pdf version of Recipe 5

This Recipe is the second in a series of three Recipes dealing with text fields. In *Recipe 4*, you implemented sheets to deal with invalid text field entries. Here, you will learn about formatters, which, among other things, can filter text on the fly as it is typed. Among other benefits, this can make it unnecessary to present a sheet to prevent entry of invalid data after the user has signaled that data entry is complete.

You already had a brief introduction to formatters in *Recipe 3, Step 2*, where you used Interface Builder to attach a built-in Cocoa formatter to the Speed Limiter text field. In this *Recipe 5*, you will learn how to write code in Project Builder to implement more specialized custom formatters.

Before turning to *Step 1*, you should prepare your project files for *Recipe 5*. Follow the same procedures you followed in the introduction to *Recipe 3*, being a subset of those outlined in *Recipe 2, Step 1*.

Screenshot 5-1: The Vermont Recipes 5 application



Vermont Recipes
http://www.stepwise.com/Articles/VermontRecipes/recipe05/recipe05.html

Introduction > Contents > Recipe 5 < BACK | NEXT >

Vermont Recipes—A Cocoa Cookbook for Mac OS X

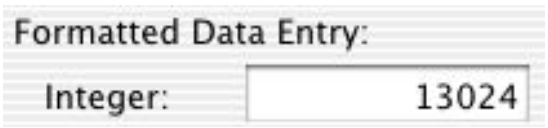
By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 5: User controls—Text fields (formatters)

Step 1: On-the-fly input filtering for integers

- Highlights:
  - Memory management for instance variables that hold value objects
  - Writing a custom formatter by subclassing NSNumberFormatter
  - Limiting typing in a text field to a specific set of characters (filtering for numeric digits)
  - Setting the tab order among text fields in a window or pane
  - Preventing tabbing among text fields from registering with the undo manager



In *Recipe 4, Step 1*, you implemented NSControl's `control:didFailToFormatString:errorDescription:` delegate method to intercept and validate an entry in a text field after the user pressed the Enter key or indicated in some other fashion that data entry in that field was complete. Validation upon completion is the usual form of data validation; it is appropriate for any data that cannot be validated until the entry is completed. For example, a field that is restricted to a limited set of values may be hard to validate before the user signals that data entry is done. Cocoa provides two very flexible built-in formatters for the most common cases, NSNumberFormatter for formatting and validating numbers and NSDateFormatter for formatting and validating dates, and Interface Builder can be used to set both of them up with a variety of common number and date formats.

Another frequently-used validation technique will be developed in this Recipe: on-the-fly text filtering and formatting. In this case, the entry is examined as the user types each character, without waiting for the Enter key or any other signal that the user has nothing more to type. This form of validation is appropriate for data that can only be composed of a limited set of characters or that has a rigid format, such as a telephone number. Illegal characters in specific positions can simply be ignored when they are typed, or a beep or other warning can be presented to the user. It is even possible to examine all of the characters that have already been typed, every time a new character is typed, in order to validate the developing value or form of the data.

In this manner, for example, an out-of-range entry can be intercepted as soon as the value of the data up to and including the last character that was typed falls outside the permissible range, and delimiter characters can be inserted automatically in the proper positions without the user's having to type them.

The built-in Cocoa formatters do not provide on-the-fly filtering or formatting. You must subclass a Cocoa formatter to provide this capability.

In this *Step 1*, you will implement simple on-the-fly input filtering, without any on-the-fly formatting, for the first of three text fields. This first field will let the user type only integer values using the numeric digits "1" through "9"; typing any other character will cause the computer to beep at the user, and the illegal character will not be displayed in the text field. You will implement a more complex formatter for the second field in *Step 2*, where you will let the user type positive decimal values using the numeric digits plus a localized decimal point; typing any other character will cause the computer not only to beep but also to present a sheet explaining the input error. The use of a sheet in response to an unacceptable character in the second field may seem overly intrusive to a fast typist, but the Aqua Human Interface Guidelines suggest doing so, and you should learn how to do it, in any event. In *Step 2*, you will also provide formatting on the fly, namely, the automatic insertion of localized thousands delimiters. The subclasses on NSNumberFormatter that you will create in *Steps 1* and *2* will be relatively simple, because inherited methods of NSNumberFormatter itself will provide code for all but one of the methods that a custom formatter must implement. In *Step 3*, you will progress into slightly more difficult territory, creating a subclass of NSFormatter to do on-the-fly and final formatting of a telephone number, complete with automatic insertion of standard North American phone number delimiters (open and close parentheses, spaces and hyphens).

You will create these formatters by implementing a delegate method provided in NSControl specifically for the purpose of on-the-fly filtering and formatting, the `control:didFailToValidatePartialString:errorDescription:` method. To trigger this delegate method, you will have to create a subclass of one of the Cocoa formatters—in *Step 1*, NSNumberFormatter—and override one of its methods. This is a somewhat complex subject and may seem to be taking you far from the subject of text fields. The excursion is a useful preface to implementing various kinds of text fields, however. It is very common for an application to have many text fields for entry of values having specific formatting requirements, and filtering and formatting such fields while the user types is extremely important to the usability of such an application. On-the-fly filtering and formatting is something that a polished application simply can't do without.

In instructions 1 and 2 you will set up the text fields that are used in all three Steps of this Recipe. There is little new here, except that in instruction 2 you will learn how to handle memory management issues when an instance variable holds a value object rather than the simple C data types that you have used up to this point. Instructions 3 and following will take you into the new subject of creating a custom formatter.

1. Use Interface Builder to create four static text fields and three new editable text fields as a group in the Text Fields tab view item. You will create all three editable fields now, although implementing formatters for the second and third fields will be left to *Steps 2* and *3*, respectively.

a. Select the Text Fields tab view item by double-clicking in the tab view area then clicking once on the Text Fields tab.

b. Drag Message Text from the Views palette to the upper left corner of the Text Fields pane, using the Aqua guides to position it, and change the text to **`Formatted Data Entry:`** as the group label. Choose Layout > Size to Fit, if necessary to make all the text visible. Use the NSTextField Info palette to uncheck the Editable and Selectable checkboxes in the Options area, since this is to be a static text field, and check the Enabled checkbox, as necessary.

c. Repeat the process thrice to create three text field labels aligned to the left vertically beneath the title you just created. Type so that the labels read **`Integer:`**, **`Decimal:`** and **`Telephone:`**, respectively.

d. Drag NSTextField icons from the Views palette thrice and align the text baseline of one of them to the right of each of the labels you just created. Make sure they are sized identically and aligned with one another vertically. This requires using the Aqua guide associated with the longest of the three static text fields to position the editable text fields properly to the right of their labels. You can temporarily type a full telephone number into the Title field of the NSTextField Info palette, using a wide digit like "8", to help size the text fields.

e. With each of the text fields you just created selected in turn, click the right alignment button in the NSTextField Info palette, so that digits typed into each text field will fill from right to left and remain aligned against the right edge of the box. Also check the Editable and Selectable checkboxes in the Options area, since these two text fields are to be editable, and check the Enabled checkbox, as necessary.

f. Choose Layout > Group in Box, give the box no title and no border, and choose Layout > Size to Fit. Now the new items can be dragged as a group without selecting all of them.

g. Drag to position the group box as necessary so that it complies with human interface guidelines and is legible.

2. Now turn to Project Builder to code the editable text fields. You must run through the checklist of tasks that you have used repeatedly to implement user controls, as originally developed in *Recipe 2, Step 3*. This should be almost second nature to you by now, but we will give a bare-bones exposition of the code here for all three of the new editable text fields at once.

   There is one new feature that is introduced in this Step, which requires that you perform some of these tasks a little differently than you did when setting up all of the previous fields in the Vermont Recipes application. For the first time, you will create an instance variable that holds an object, rather than a variable holding one of the standard C data types that you have worked with to this point. The new Telephone text field will display a telephone number, the data for which will be held in an instance variable of type NSString, an object, in the MySettings model object. An instance variable that holds an object requires special handling, including allocating and initializing the new data object when a new window is created, deallocating the object when the window is closed, and attending to memory management while the new object is in use. Each of these points is discussed at instructions

2.b., d., h., and i., below. Later, in *Recipe 6*, you will learn to use other value objects, such as NSNumber objects, instead of C data types for many text fields.

a. **User control outlet variables and accessors**. In the header file `MyWindowController.h`, declare three outlet variables after the existing variable declarations, as follows:

```
// Formatted text fields
IBOutlet NSTextField *integerTextField;
IBOutlet NSTextField *decimalTextField;
IBOutlet NSTextField *telephoneTextField;
```

Still in `MyWindowController.h`, also declare an accessor method for each outlet at the end of the Accessor methods and conveniences section, as follows:

```
// Formatted text fields
- (NSTextField *)integerTextField;
- (NSTextField *)decimalTextField;
- (NSTextField *)telephoneTextField;
```

Turn to the source file `MyWindowController.m` and define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Formatted text fields

- (NSTextField *)integerTextField {
    return integerTextField;
}

- (NSTextField *)decimalTextField {
    return decimalTextField;
}

- (NSTextField *)telephoneTextField {
return telephoneTextField;
}
```

b. **Data variables and accessors**. In the header file `MySettings.h`, declare three new variables after the existing variable declarations, as shown below. They hold an `int`, a `float` and an `NSString`, respectively. Note that the second is typed as a float for convenience, even though the second text field will be displayed as a decimal value without any of the additional exponent formatting characteristic of true floating point values. In *Recipe 6*, you will be introduced to Cocoa's NSDecimalNumber class, which may be more appropriate in a typical Cocoa programming project.

```
// Formatted values
int integerValue;
float decimalValue;
NSString *telephoneValue;
```

In `MySettings.h`, also declare the corresponding accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Formatted values

- (void)setIntegerValue:(int)value;
- (int)integerValue;

- (void)setDecimalValue:(float)value;
- (float)decimalValue;

- (void)setTelephoneValue:(NSString *)value;
- (NSString *)telephoneValue;
```

In the source file `MySettings.m`, define the accessor methods at the end of the Accessor methods and conveniences section, as follows:

```
// Formatted values

- (void)setIntegerValue:(int)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setIntegerValue:integerValue];
    integerValue = value;
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRIntegerValueChangedNotification
object:self];
}

- (int)integerValue {
    return integerValue;
}

- (void)setDecimalValue:(float)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setDecimalValue:decimalValue];
    decimalValue = value;
```

```
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRDecimalValueChangedNotification
object:self];
}

- (float)decimalValue {
    return decimalValue;
}

- (void)setTelephoneValue:(NSString *)value {
    [[[self undoManager]
prepareWithInvocationTarget:self]
setTelephoneValue:telephoneValue];
    [telephoneValue autorelease];
    telephoneValue = [value copy];
    [[NSNotificationCenter defaultCenter]
postNotificationName:VRTelephoneValueChangedNotification
object:self];
}

- (NSString *)telephoneValue {
    return telephoneValue;
}
```

Take a close look at the `setTelephoneValue:` accessor method, above, because we slipped in a very important new lesson. Until now, all of the values you have used in MySettings have been standard C data types. Now, for the first time, you have used an object, instead. This requires careful consideration of memory management issues.

First, when this accessor method is called to install a pointer to the incoming string object into the instance variable, there will already exist an old string object to which the variable currently refers. Even in the case of a newly created window, there will be a string object in `telephoneValue`, because the MySettings object's designated initializer will have allocated and initialized one and assigned it to the instance variable, as described in instruction 2.h., below. This is your last chance to ensure that the old object gets released before the variable is set to the new object. Once the variable has been changed to refer to the incoming object, there will be no way to access the old object to release it, and your application will suffer a memory leak. You must therefore ensure that the old object gets released before assigning a new object to the instance variable. If you're very careful, you might be able simply to release the old object. In general, however, despite the increased overhead and possible increased difficulty of debugging, it is better to autorelease the old object in these circumstances. Clients will have had access to the instance variable and may have created their own references to it, leaving them dependent on the old object's remaining valid during the current event cycle.

Second, you shouldn't simply assign the incoming object to the instance variable and retain it, because these other, shared references to the incoming object may exist. If you simply assign the incoming object to the instance variable and retain it, it is possible that one of those shared references will change the value of the object out from under you, without your knowledge. In general, in the case of a so-called "value object" like `telephoneValue`, you should make a copy of the incoming object instead of retaining it, so that it will take on an independent existence under your exclusive control without risk of being altered by other, shared objects. (So-called "entity objects"—objects that represent complex systems rather than relatively simple values like a telephone number—are usually treated differently, but you needn't bother with those for now.)

c. **Notification variables**. Return to the header file `MySettings.h`, at the bottom of the file, to declare the notification variables, as follows:

```
// Formatted values
extern NSString *VRIntegerValueChangedNotification;
extern NSString *VRDecimalValueChangedNotification;
extern NSString
*VRTelephoneValueChangedNotification;
```

Turn back to the source file `MySettings.m`, near the top of the file, to define the notification variables, as follows:

```
// Formatted values
NSString *VRIntegerValueChangedNotification =
@"IntegerValue Changed Notification";
NSString *VRDecimalValueChangedNotification =
@"DecimalValue Changed Notification";
NSString *VRTelephoneValueChangedNotification =
@"TelephoneValue Changed Notification";
```

d. **GUI update methods**. In the header file `MyWindowController.h`, declare user interface update methods at the end of the Specific view updaters section, as follows:

```
// Formatted text fields
- (void)updateIntegerTextField:(NSNotification
*)notification;
- (void)updateDecimalTextField:(NSNotification
*)notification;
- (void)updateTelephoneTextField:(NSNotification
*)notification;
```

In the source file `MyWindowController.m`, define these specific update methods at the end of the Specific view updaters section, as follows:

```
// Formatted text fields

- (void)updateIntegerTextField:(NSNotification
*)notification {
    [self updateTextField:[self integerTextField]
setting:[NSString localizedStringWithFormat:@"%d",
[[self mySettings] integerValue]]];
}

- (void)updateDecimalTextField:(NSNotification
*)notification {
    [self updateTextField:[self decimalTextField]
setting:[NSString localizedStringWithFormat:@"%f",
[[self mySettings] decimalValue]]];
}

- (void)updateTelephoneTextField:(NSNotification
*)notification {
    [self updateTextField:[self telephoneTextField]
setting:[[self mySettings] telephoneValue]];
}
```

Notice that the last of the update methods did not need to call NSString's `localizedStringWithFormat:` method, not only because it refers to U.S. phone numbers only, which don't require localization because they should appear the same to people calling the U.S. from anywhere in the world, but also because the value was already a string. It can be passed directly to the `updateTextField:setting:` method without conversion.

e. **Notification observer**. Register the window controller as an observer of the notifications that will trigger the updater methods, by inserting the following statement in the `registerNotificationObservers` method of the source file `MyWindowController.m`, after the existing registrations:

```
// Formatted text fields
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateIntegerTextField:)
name:VRIntegerValueChangedNotification object:[self
mySettings]];
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateDecimalTextField:)
name:VRDecimalValueChangedNotification object:[self
mySettings]];
[[NSNotificationCenter defaultCenter]
addObserver:self
selector:@selector(updateTelephoneTextField:)
name:VRTelephoneValueChangedNotification
object:[self mySettings]];
}
```

f. **Action methods**. In the header file `MyWindowController.h`, at the end of the Action methods section, add the following:

```
// Formatted text fields
- (IBAction)integerTextFieldAction:(id)sender;
- (IBAction)decimalTextFieldAction:(id)sender;
- (IBAction)telephoneTextFieldAction:(id)sender;
```

In the source file `MyWindowController.m`, define the action methods at the end of the Action methods section, as follows:

```
// Formatted text fields

- (IBAction)integerTextFieldAction:(id)sender {
    [[self mySettings] setIntegerValue:[sender
intValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Integer
Value", @"Name of undo/redo menu item after Integer
text field was set")];
}

- (IBAction)decimalTextFieldAction:(id)sender {
    [[self mySettings] setDecimalValue:[sender
floatValue]];
```

```
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Decimal
Value", @"Name of undo/redo menu item after Decimal
text field was set")];
}

- (IBAction)telephoneTextFieldAction:(id)sender {
    [[self mySettings] setTelephoneValue:[sender
stringValue]];
    [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Telephone
Value", @"Name of undo/redo menu item after
Telephone text field was set")];
}
```

g. **Localizable.strings**. Update the `Localizable.strings` file by adding "Set Integer Value," "Set Decimal Value," and "Set Telephone Value":

```
/* Formatted text fields */
/* Name of undo/redo menu item after Integer text
field was set */
"Set Integer Value" = "Set Integer Value";
/* Name of undo/redo menu item after Decimal text
field was set */
"Set Decimal Value" = "Set Decimal Value";
/* Name of undo/redo menu item after Telephone text
field was set */
"Set Telephone Value" = "Set Telephone Value";
```

h. **Initialization**. Leave the first two data variables uninitialized. Objective-C will initialize them to zero.

   However, in the case of the `telephoneValue` variable, you have to consider where the initial NSString object will come from when a new window is created. The answer is that you must instantiate it yourself, typically in the model object's designated initializer. Go to the `initWithDocument:` method in `MySettings.m` and add the following line before the call to the undoManager's `enableUndoRegistration` method:

```
(NSString *)telephoneValue = @"(800) 555-1212";
```

   If you had wanted to leave the `telephoneValue` variable empty, instead of providing a default value, you still would have had to allocate and initialize a blank string object. You can allocate, initalize, and assign an empty string object in Cocoa all at once, like this:

```
(NSString *)telephoneValue = @"";
```

Alternatively, you can do it like this, but this is implemented internally in the same way as the first version in most circumstances:

```
(NSString *)telephoneValue = [[NSString alloc]
init];
```

Or simply this:

```
(NSString *)telephoneValue = [NSString string];
```

Finally, you must release every object that you allocate, so add a `dealloc` override method to `MySettings.m`, after the designated initializer, as shown below. It will deallocate whatever object is currently referred to by the instance variable, including the object that was assigned by the last call to the `setTelephoneValue:` accessor method described in instruction 2.b., above, if the user has typed a new value into the field.

```
- (void)dealloc {
    [[self telephoneValue] release];
    [super dealloc];
}
```

i. **Data storage**. In the source file `MySettings.m`, define the following keys in the Keys and values for dictionary section:

```
// Formatted values
static NSString *integerValueKey = @"IntegerValue";
static NSString *decimalValueKey = @"DecimalValue";
static NSString *telephoneValueKey =
@"TelephoneValue";
```

Immediately after that, add these lines at the end of the `convertToDictionary:` method:

```
// Formatted values
[dictionary setObject:[NSString
stringWithFormat:@"%d", [self integerValue]]
forKey:integerValueKey];
[dictionary setObject:[NSString
stringWithFormat:@"%f", [self decimalValue]]
forKey:decimalValueKey];
[dictionary setObject:[self telephoneValue]
forKey:telephoneValueKey];
```

Again, the `telephoneValue` variable does not require conversion to a string because it is already a string.

And add these lines near the end of the `restoreFromDictionary:` method, before the call to `[[self undoManager] enableUndoRegistration]`:

```
// Formatted values
[self setIntegerValue:[[dictionary
objectForKey:integerValueKey] intValue]];
[self setDecimalValue:[[dictionary
objectForKey:decimalValueKey] floatValue]];
[self setTelephoneValue:[dictionary
objectForKey:telephoneValueKey]];
```

Again, the parameter passed to `setTelephoneValue:` does not have to be converted to a string because it is already a string.

Notice that the values of the parameters passed to `setIntegerValue:` and the other two accessor methods are not cast to `int`, `float`, or `NSString*`. Somehow, we left some casts in the equivalent calls to `setPersonalityValue:`, `setSpeedValue:` and `setQuantumValue:` in the code files prepared in earlier Recipes. Why don't you remove them now, if they are in your copies.

j. **GUI update method invocation**. Finally, in `MyWindowController.m`, add the following calls at the end of the `updateWindow` method :

```
// Formatted text fields
[self updateIntegerTextField:nil];
[self updateDecimalTextField:nil];
[self updateTelephoneTextField:nil];
```

3. Now you are ready to implement on-the-fly input filtering for the first of the three new text fields

(you will defer the code for filtering and formatting the other two fields to *Steps 2* and *3*). First, in `MyWindowController.m`, add the following delegate method in the Formatter errors section following the `control:didFailToFormatString:errorDescription:` method. It is a delegate method and doesn't require a declaration.

```
- (void)control:(NSControl *)control
didFailToValidatePartialString:(NSString *)string
errorDescription:(NSString *)error {
    if (control == [self integerTextField]) {
        NSBeep();
    }
}
```

As you have seen in other contexts, you must begin by checking whether it was the control you are interested in that caused the delegate method to be called. This is because other controls may also appoint MyWindowController as their delegate, and they may also invoke on-the-fly input filters. This delegate method will be triggered by filtering errors in all of them. You therefore have to be careful to differentiate among the controls if, as will be the case here, you want to provide different responses to filtering errors in different controls. For Integer text field filtering errors, the application will simply beep, but in *Step 2* you will cause the application to persent a sheet in response to filtering errors in the Decimal text field.

4. Remember that, because this is a delegate method, it will be called automatically by NSControl because you have implemented it in MyWindowController—but only if you remember to appoint MyWindowController as the delegate of this control. You had best get that out of the way right now, because it is easy to forget. In Interface Builder, Control-drag from the Integer text field to the File's Owner icon, which you recall is a proxy for the MyWindowController object, then connect the `delegate` target.

5. The next task is to create the new filter and attach it to the Integer text field. You should read the NSControl and NSFormatter class reference documents now to understand the background for the code you are about to write. In summary, if NSControl's `control:didFailToValidatePartialString:errorDescription:` method is implemented in a control's delegate, then, every time the user types a character, Cocoa will automatically format and validate the text field by calling the `isPartialStringValid:newEditingString:errorDescription:` method of any formatter object that has been attached to the control and that implements the method. You have already satisfied some of these conditions by implementing the `control:didFailToValidatePartialString:errorDescription:` method in MyWindowController and appointing MyWindowController as the control's delegate. All you have to do now, therefore, is to write an appropriate formatter and attach it to the Integer text field.

   In *Recipe 3, Step 2*, you performed these last remaining steps using InterfaceBuilder, making use of its built-in facilities for instantiating an NSNumberFormatter object and connecting it to a user

control. Here, however, you will perform these steps programmatically. You will do this backwards, leaving the best part—writing the custom formatter subclass—to last in instruction 6., below.

a. The formatter will be created as a separate object in its own header and source files. Let's say that the class will be named MyIntegerFilter, and that it will be declared as a subclass of NSNumberFormatter in order to inherit all the methods and number-handling capabilities of that built-in Cocoa class. The only method you will have to implement in the subclass is the `isPartialStringValid:newEditingString:errorDescription:` method.

   With those points in mind, you can write a method in MyWindowController that will instantiate an instance of the MyIntegerFilter class and call it in MyWindowController's w`indowDidLoad` method.

   In `MyWindowController.h`, before the Formatter errors subsection of the Input validation and formatting methods section, declare the method that will instantiate a MyIntegerFilter object and attach it to the Integer text field, as follows:

```
// Formatter constructors

- (void)makeIntegerFilter {
    MyIntegerFilter *integerFilter =
[[MyIntegerFilter alloc] init];
    [integerFilter setFormat:@"##0"];
    [[[self integerTextField] cell]
setFormatter:integerFilter];
}
```

   This is very straightforward. You use the now-familiar standard means to allocate and initialize an instance of the MyIntegerFilter class. Since it will be a subclass of NSNumberFormatter, you know that you can call the `setFormat:` method inherited from NSNumberFormatter to define a template for the appearance of the integers that will be entered in the Integer text field. Finally, you call the `setFormatter:` method, which is built into every cell-based user control, to attach the new formatter to the text field.

   Note that no range checking is done to ensure that the value typed can safely be converted to an integer data type internally; that is an exercize for another day.

   You will worry about reclaiming the memory set aside for the formatter in a few moments.

   Declare this method in `MyWindowController.h` before the Formatter errors subsection of the Input validation and formatting methods section:

```
// Formatter constructors
- (void)makeIntegerFilter;
```

Also, because the implementation of the method calls methods of the MyIntegerFilter class, you must import that class into `MyWindowController.m` by adding the following line to the end of the import directives at the top of the file:

```
#import "MyIntegerFilter.h"
```

a. In anticipation of writing several custom formatters for the application, you might want to follow your previous practice of writing an umbrella method where you can collect all of your calls to similar methods, as you did when you wrote the `registerNotificationObservers` and `updateWindow` methods in earlier Recipes. Create a method to collect formatter constructors like `makeIntegerFilter` and call it `makeFormatters`. In `MyWindowController.m`, after the `updateWindow` method, insert the following:

```
- (void)makeFormatters {
    [self makeIntegerFilter];
}
```

Declare it in `MyWindowController.h`, after the `updateWindow` method:

```
- (void)makeFormatters;
```

a. A text field's formatter can be instantiated at the same time its window is created, so add the following line to the `windowDidLoad` method in `MyWindowController.m`, before the call to `registerNotificationObservers`:

```
[self makeFormatters];
```

b. The NSNumberFormatter class reference document suggests that you autorelease a custom formatter when you create it, but you did not do this in instruction 5.a., above. You are responsible for autoreleasing or releasing every object that you allocate yourself, so you must deal with this issue somewhere. In general, it is preferable to release an object explicitly, if you can, in order to avoid the performance hit associated with excessive use of the autorelease pool.

Since you know that this formatter must remain associated with the Integer text field for the life of the window, you can safely wait until the window is closed to release it. Add the following line to the `dealloc` method in `MyWindowController.m`, just before the call to super's `dealloc` method. It calls NSControl's built-in `formatter` accessor method to locate the formatter attached to the Integer text field.

```
[[[self integerTextField] formatter] release];
```

This setup should work correctly even if, in some future version of Vermont Recipes, you should decide to attach the same formatter to multiple text fields in the same window. You won't have to instantiate new copies of the formatter for the new text fields, because the formatter can only be placed in use by one text field at a time, namely, the field that has keyboard focus.

6. You are now ready to write the new formatter. It is very simple, after all that preparation.

    a. Using Project Builder techniques that you have now used several times, create new Cocoa header and source files and name them **`MyIntegerFilter.h`** and **`MyIntegerFilter.m`**, respectively. In the Groups & Files pane of the Project Builder window, create a new subgroup in the Classes group and name it **`Formatters`**. Then choose Project > Add Files…, add the two new files, and drag their icons into the new Formatters group in the Groups & Files pane of the Project Builder window, if necessary.

    b. Set up the header file `MyIntegerFilter.h` so that it imports the Cocoa umbrella framework and inherits from NSNumberFormatter. The skeleton should look like this:

```
#import <Cocoa/Cocoa.h>

@interface MyIntegerFilter : NSNumberFormatter {
}

@end
```

    c. Set up the source file `MyIntegerFilter.m` so that it imports its header file and provide the implementation skeleton, as follows:

```
#import "MyIntegerFilter.h"

@implementation MyIntegerFilter

@end
```

    d. Finally, add the following definition of the filter method, as a tentative first stab at it (you will shortly abandon this initial approach and adopt a better one):

```
// Input filter

- (BOOL)isPartialStringValid:(NSString
*)partialString newEditingString:(NSString
**)newString errorDescription:(NSString **)error {
    if (!([[NSCharacterSet decimalDigitCharacterSet]
characterIsMember:[partialString
characterAtIndex:[partialString length] - 1]])) {
        *newString = nil;
        *error = NSLocalizedString(@"Input is not an
integer", @"Presented when user value not a numeric
digit");
        return NO;
    }
    return YES;
}
```

As an override method, this does not require a declaration. In fact, for this formatter, the header file will be empty.

On entry, the `partialString` parameter value is the string as the user typed it in the text field, including the most recent character that was typed—which may be an illegal character. You obtain the last character by using NSString's `characterAtIndex:` method, using the index of the last character in the field, `[partialString length] - 1`. (You may notice that this is a mistake, because the user might have started typing somewhere other than at the end of the text field; but we start here for simplicity.) The `newEditingString` parameter contains a value that you can pass back to the calling routine, which you may edit within the method before returning it by reference. The `ErrorDescription` parameter allows you to return to the caller, by reference, a string object containing a displayable description of any error discovered when the last character was typed. The `error` value should, of course, be localized.

To test the legality of the last character typed, you check it for membership in the `decimalDigitCharacterSet` character set, using NSCharacterSet's `characterIsMember:` method. The `decimalDigitCharacterSet` method returns one of several character sets that are already created in Cocoa's NSCharacterSet class. This character set consists solely of the characters "1" through "9," which is just what you want. (Note that you haven't tested whether `partialString` contains any characters. Assume for the moment that this method is only invoked by the typing of a printable character, and that `partialString` will therefore always have a length greater than zero.You will deal with the Delete key and the Forward Delete key later.)

If the last character typed is determined to be invalid because it is not a member of the character set, you set `newString` to `nil` and return it by reference, and you define an error

description string that you also return by reference. You then return `NO` as the method result to indicate that `partialString` is unacceptable, so that Cocoa will not display the last character in the text field. If `newString` is `nil` when the method returns `NO`, Cocoa will display the previous contents of the text field without the last character that was typed. Although you are not making use of the capability to return an edited string here, you could do so and it would be displayed in the text field in place of the string the user typed. You define an error string, just in case you might redesign the application later, but in this version you will not make use of it but instead only sound a beep. If the last character passes the validity test, you simply return `YES` to indicate that `partialString` is acceptable, and Cocoa will go ahead and display it in the text field as the user typed it, including the last character.

If you were to hook up everything in the nib file and build and run the application now, you would discover that the formatter works—sort of. If you start typing into the Integer text field after selecting its entire existing contents, any illegal characters are indeed rejected with a warning beep, as you expected. However, if you place the insertion point at the front or in the middle of the existing contents of the text field before you start typing, illegal characters are accepted without complaint. The reason is immediately apparent: your formatter explicitly tests the last character in the entire field—that is, the character at the end—not the character most recently typed. As you have just discovered, they are not always the same.

e. Until recently, the only available approach to a solution would have been to evaluate the entire string for illegal characters, and it would have been difficult to track the insertion point. However, when you read the NSFormatter class reference document and the Foundation release notes for Mac OS X 10.0, you find that a new, enhanced version of the method has been made available, and it seems to offer a much better approach. Among other things, it keeps track of the current location of the insertion point and so allows you to track the most recent printable character that was typed, even if it is not at the end of the text field. In addition, it gives you the original string, before any new text was inserted, and the original location and length of the insertion point or selection. It also offers enhanced facilities for editing the string and tracking or altering the user's selection, but all you need here is the location of the insertion point before and after the user typed or pasted new characters into the text field.

   One way to attempt to deal with the problem presented by your last approach might be to delete the method you wrote in instruction 6.d., above, in `MyIntegerFilter.m`, and substitute the following, which uses the new NSFormatter method:

```
- (BOOL)isPartialStringValid:(NSString
**)partialStringPtr
proposedSelectedRange:(NSRangePointer)proposedSelRangePtr
originalString:(NSString *)origString
originalSelectedRange:(NSRange)origSelRange
errorDescription:(NSString **)error {
    if (!([[NSCharacterSet decimalDigitCharacterSet]
characterIsMember:[*partialStringPtr
characterAtIndex:origSelRange.location]])) {
        *error = NSLocalizedString(@"Input is not an
integer", @"Presented when user value not a numeric
digit");
        return NO;
    }
    return YES;
}
```

This solves the problem of typing from an insertion point or selection at the beginning or in the middle of the text field, by making use of the new method's `origSelRange` parameter and its `location` element to provide the index for the `characterIsMember:` method. You know that any new character typed appears at the original insertion point, or at the beginning of the original selection that was replaced by the new character, and the `origSelRange` parameter makes this information available to you.

As you see, the new method's parameters are considerably more complex than those of the old method, tracking both the original string before another character was typed and the new string after the character was typed, as well as tracking the selection range both before and after typing. The selection range is used, rather than the location of the insertion point, because the user may have selected a range of characters to be replaced by a newly typed character. A formatter may need to use the contents of the selection, and it will at least need to know where the selection began. Furthermore, it allows you to alter the selection range if you choose to edit the user's entry before returning it to the caller. Finally, instead of giving you the new string in one parameter and allowing you to return an edited string by reference in another parameter, it gives you two strings and allows you to return an edited string by reference in one of them. You will exercise these new facilities more fully when you write more complex formatters in *Steps 2 and 3*.

Here, however, you don't need to monkey with most of the new parameters, since you only care about the location of the insertion point. The `originalSelectedRange` parameter is of type `NSRange`; this is a C structure, declared in `NSRange.h`, consisting of the `location` and `length` of a range of characters within a string. Its use in Cocoa, including the dot notation familiar to C programmers for accessing individual elements of a structure, is one of the reasons why newcomers to Cocoa development are urged to learn standard C as

well as the Objective-C extensions. The location of the user's selection in the text field just before typing another character is the location of the insertion point, whether the user had selected a range of characters for replacement or merely placed the insertion point with a mouse click. This is precisely the value you need in order to obtain that new character, wherever in the string it might have been typed, and to test it for membership in `decimalDigitCharacterSet`.

This new method actually makes your implementation simpler than it was before. Instead of returning a `nil` value in the old edited string parameter, you just leave the new two-way partial string parameter undisturbed. By returning `NO` as the method result, Cocoa knows that this means you wish to reject the newly-typed character. The only other change in your implementation, besides using `origSelRange.location` instead of the index of the last character of the new string, is to obtain the user's proposed new string by dereferencing the new `partialStringPtr` parameter instead of getting the old `partialString` parameter.

f. Unfortunately, the need to accommodate typing at the beginning or in the middle of the text field was not the only problem with your initial approach. In addition, if the user were to paste a string into the field and the string contained invalid characters in any of several positions, they might also be accepted without generating an error. This is because the formatter tests only a single new input character, overlooking the fact that whole strings can be pasted into a text field. You need to fix this problem, too.

   The bottom line is that any formatter designed to filter out illegal characters must test a string, rather than testing a single character, because the user might at any time paste in a string of characters. It turns out to be easy to accommodate this need, altering the code you have already developed only as much as required to test an input string instead of an input character. This requires only two new code snippets, one to obtain the user's input of one or more characters in the form of a string, and another to test the string, instead of testing a character, for membership in the filtering character set. Delete the initial attempt immediately above, and substitute the following in `MyIntegerFilter.m`:

```
- (BOOL)isPartialStringValid:(NSString
**)partialStringPtr
proposedSelectedRange:(NSRangePointer)proposedSelRangePtr
originalString:(NSString *)origString
originalSelectedRange:(NSRange)origSelRange
errorDescription:(NSString **)error {
    if ([[*partialStringPtr
substringWithRange:NSMakeRange(origSelRange.location,
(*proposedSelRangePtr).location - origSelRange.location)]
rangeOfCharacterFromSet:[[NSCharacterSet
decimalDigitCharacterSet] invertedSet]
options:NSLiteralSearch].location != NSNotFound) {
```

```
        *error = NSLocalizedString(@"Input is not an
integer", @"Presented when user value not a numeric
digit");
        return NO;
    }
    return YES;
}
```

That complicated `if` test is not meant to show off how convoluted an Objective-C statement can be made to appear, but only to cram as much work as possible into a single Boolean statement. We will break it into pieces to explain what it does.

The first part of the `if` test is `[*partialStringPtr substringWithRange:NSMakeRange(origSelRange.location, (*proposedSelRangePtr).location - origSelRange.location)]`. It obtains what the user typed or pasted into the field, in the form of a string. The string may contain a single character if the user typed at the keyboard, or one or more characters if the user pasted something from the pasteboard, or nothing if the user typed the Delete key or the Forward Delete key. By getting the user's input all at once in a string, whether it was typed or pasted, you save yourself the bother of getting it one way if typed and a second way if pasted. It is very easy to do this at once using this new NSFormatter method, introduced in Mac OS X 10.0. The `partialStringPtr` parameter value contains the entire contents of the field after the user typed or pasted into it, including the new character or characters. NSString's `substringWithRange:` method is used to obtain the portion of the text field's contents that was just added by a typing or pasting operation. The range is created using the `NSMakeRange()` function, a macro that is defined in `NSRange.h` (Cocoa makes the macro available, so you use it; you could just as well have written `(NSRange){origSelRange.location, (*proposedSelRangePtr).location - origSelRange.location}`.) The `proposedSelRangePtr` parameter value contains, in its `location` element, the proposed new location of the insertion point after the user's entry, and the `location` element of the `origString` parameter value contains the original location of the insertion point before the user's entry. The difference between these two locations is the range that marks out the character or characters that were just inserted—namely, what the user typed or pasted.

Having the input string in hand, you need only filter every character in it. Since you need to detect whether any one of the characters in the string is illegal, you cannot test whether any character in a valid character set is found, but instead must test whether any character in an illegal character set is found. For this, you need the inverse of Cocoa's built-in `decimalDigitCharacterSet`; that is, a set containing every character *other than* the decimal digits. You also need a method that tests every character in a string for membership in the inverted character set. For these purposes, NSCharacterSet provides the `invertedSet` method, and NSString provides the `rangeOfCharacterFromSet:options:` method. The latter works by reporting the range of the first appearance of any character or characters in

the set. It is worth looking at the NSString reference document for an explanation of how this method works and to review the several alternatives that are available. The method returns an NSRange structure defining the `location` and `length` of the substring for which it is searching within the target string, if one is found. If the substring is not found, the location element of the structure is returned as the value of the `NSNotFound` constant, along with a `length` of `0`. This is all accomplished by the second part of the `if` test, `[<string> rangeOfCharacterFromSet:[[NSCharacterSet decimalDigitCharacterSet] invertedSet] options:NSLiteralSearch].location != NSNotFound)`.

What the `if` test boils down to, therefore, is whether the reported location of an illegal character in whatever the user typed or pasted is not equal to `NSNotFound`: `<illegal character>.location != NSNotfound`.

g. Don't forget to add the "Input is not an integer" error message to the `Localizable.strings` file:

```
/* Formatters */
/* Presented when user value not a numeric digit */
"Input is not an integer" = "Input is not an
integer"
```

h. Unfortunately, if you were to hook everything up in Interface Builder and test the formatter now, you would discover a glitch. If the user were to type some numbers into the field and then use the Delete key to delete all of them, the insertion point would disappear. It appears that this happens because of an issue in the Cocoa code that makes use of the two on-the-fly formatting methods you have explored in this Step.

There is a workaround. It isn't pretty, but it's short and it works. Insert the following block of code near the end of the `isPartialStringValid:...` method, just before it returns `YES`:

```
if ([*partialStringPtr length] == 0) {
    [[[NSApp keyWindow] fieldEditor:NO
forObject:nil]
setSelectedRange:*proposedSelRangePtr];
}
```

This block treats as a special case the situation where the text field is empty, since the Cocoa issue only arises in that circumstance. The operative statement directly sets the selection of the window's field editor to the `location` and `length` of `*proposedSelRangePtr`. Using a variety of debugging techniques, we have established that these `location` and `length` values are correct when the formatter has completed its work, and that the problem of the

disappearing insertion point therefore must reside in Cocoa code that makes use of these values to set the text field's insertion point or selection. Because this new block of code to work around the bug is inside a running formatter, you know that your text field's window is currently the application's key window and that this window's field editor is currently being used to edit your text field's contents. It is therefore safe to use the application's `keyWindow` method and NSWindow's `fieldEditor:forObject:` method to access the field editor where your text field's contents are being edited. A window's field editor is an instance of NSTextView, so you can use NSTextView's `setSelectedRange:` method to set the insertion point directly.

You will want to check future releases of Mac OS X to see whether this issue has been resolved, and remove this block of code if it has.

7. Inform the nib file of the new outlets and actions you have created in the code files for the Integer and Decimal text fields, then connect them to the two new text fields. You will also learn here an Interface Builder technique that is new to you, to set the tab order of the text fields in the Text Fields pane.

    a. In Interface Builder, select the Classes tab in the nib file window, then choose Classes > Read File…. In the resulting dialog, select the two header files in which you have created outlets or actions, `MySettings.h` and `MyWindowController.h`, then click the Parse button.

    b. Select the Instances tab and Control-drag from the File's Owner icon to each of the three new text fields in turn and click their outlet names, then click the Connect button in the Connections pane of the File's Owner Info palette.

    c. Control drag from each of the three new text fields to the File's Owner icon in turn and click the targets in the left pane and the appropriate actions in the right pane of the Outlets section of the Connections pane of the NSTextField Info palette, then click the Connect button.

    d. *First responder and next key view*. Here, you are introduced to a new procedure that must usually be followed when creating editable text fields. The Aqua Human Interface Guidelines indicate that typing in a text field should not take place until the user has deliberately selected a text field for editing. Besides selecting a text field by clicking or double-clicking in it, users are accustomed to finding that pressing the Tab key while no text field is active in a window selects the first text field (the upper left field, on Roman systems). Further, pressing the Tab key while a text field is already selected validates and commits any pending data in that field, and it causes the insertion point to jump from one to another editable text field in the window in a sensible order, making it easy to enter data in a number of text fields in succession with one's hands on the keyboard. You can and should provide for this behavior using Interface Builder in windows where it is appropriate.

    The first four instructions, below, illustrate how to use the Instances tab of the `MyDocument.nib` window for this purpose. When the Instances tab is in its default Icon View, you only see so-called "top level" object icons in the window. They represent objects

that have been instantiated in Interface Builder, or their proxies. In order to connect lower-level objects that have been installed in a top-level object—such as the Integer text field, here—you can switch to the List View of the Instances tab and tunnel down through the outline levels until you find the objects you want, then Control-drag among them or between them and objects in the main document window. In some cases, such as setting the initial first responder of a tab view item, this is the only available technique for making connections; you cannot Control-drag from the tab view item in the main document window.

i. In the `MyDocument.nib` window, select the Instances tab, then click the List View icon above the vertical scroll bar. You see an expandable outline or list view of the Instances tab which allows you to navigate through the entire hierarchy of objects represented in the nib file.

ii. Find NSWindow at the bottom of the list and expand it, then find NSTabView and expand it. You may have to scroll down in the window to see everything. This exposes the three tab view items you have created. If you expand any of the tab view items, you will see the hierarchy of user controls contained in that pane, including the group boxes that you have created in some of them to hold other controls. The outline in the List View reflects the nesting hierarchy of the application's main document window, its tab view, its tab view items, and their controls.

iii. Control-drag from NSTabViewItem (Text Fields) to the Integer text field in the main document window. You could instead have dragged to the Integer text field in the list itself, but it is harder to know which text field is which in the List View because all three currently have values of zero.

iv. Click the `initialFirstResponder` outlet in the NSTabViewItem Info palette, then click the Connect button. You have now made the Integer text field the initial first responder for the Text Field pane of the window. As a result, when the user presses the Tab key after selecting the Text Fields pane, the Integer text field will be selected and the user can type over whatever value it currently contains.

v. Now, in the main document window, Control-drag from the Integer text field to the Decimal text field.

vi. Click the `nextKeyView` outlet in the NSTextField Info palette and click the Connect button. You have now made the Decimal text field the second key view in the pane, and the user can tab directly from the Integer text field to the Decimal text field. Do the same thing to enable the user to tab from the Decimal text field to the Telephone text field.

vii. You normally make the first text field in a window or pane the next key view of the last text field in the window or pane, so that the user's tabbing will continue circling through all the text fields in order. You haven't finished adding text fields to this pane, yet, but go ahead and do it anyway, in order to see how it works. Control-drag from the Telephone text field to the Integer text field, select the `nextKeyView` outlet, and

click Connect. (You will have to remember to change this in a later Step, after you add another text field to this pane.)

viii. You may recall that you did not make the Speed Limiter text field the initial first responder of the Sliders pane. Why don't you go back and take care of that now, using the technique outlined in instructions 7.d.i.–iv., above. You won't need to deal with the `nextKeyView` outlet because there is only the one text field in that pane. Now, pressing tab upon selecting the Sliders pane will select this text field. The Buttons pane contains no text fields, so it doesn't need an initial first responder.

ix. Generally, you make the first editable text field in a window the window's initial first responder. However, since this window has no editable text fields, except for the text fields in two of the tab view items, there is no point in doing this. Way back in *Recipe 1, Step 2.6.3*, you made the Checkbox control in the Buttons pane the window's initial first responder, just to see how it is done. Go back now and disconnect that connection, since you now know it is inappropriate. As a result, the default initial first responder of the window will be the window itself. Once the user clicks on anything else in the window, Cocoa will begin updating the current first responder automatically.

Note that it is possible to connect the next key views of all controls, not just text fields. For example, a form (a matrix of text fields) can be a next key view. (In OpenStep, you could even use this technique to let the user tab in order among buttons and other non-text user controls. This was necessary so that OpenStep applications could behave as good citizens when running under Windows. Cocoa does not run under Windows, and I understand that the ability to tab to a button has been disabled in current versions of Cocoa.)

8. If you were to build and run the application now, you would discover one last bunch of problems, all related to one another and easily fixed. When you tab from text field to text field in the Text Fields pane, the window is "dirtied"—a dot appears in the close button in the window's title bar, indicating that a change has been made to the data displayed in the window. In addition, the Undo menu item in the Edit menu now indicates, for example, that you may "Undo Set Integer Value" or "Undo Set Decimal Value," and so on. Yet you haven't changed the value of any of the text fields in the pane, but only tabbed through all of them. What is wrong?

   If you set a few strategic breakpoints in the code and run the application in debug mode, you will discover that tabbing past a text field invokes its action method. The action method sets the data value associated with the text field—in this case, resetting it to the same value it already holds. It does this by invoking the "set…" accessor method, which invokes the undo manager; and when the accessor method returns, the action method also sets the Undo menu item name. None of this is appropriate when you merely tab through a text field without changing its value.

   An obvious way to prevent this behavior is to begin every text field's action method with a test, checking whether the value currently displayed in the user control is not the same as the value currently held in the associated data variable in the MySettings model object. If they are the same, there is no need to invoke the action method. In `MyWindowController.m`, revise the three text

field action methods to read as follows:

```
- (IBAction)integerTextFieldAction:(id)sender {
    if ([[self mySettings] integerValue] != [sender
intValue]) {
        [[self mySettings] setIntegerValue:[sender
intValue]];
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Integer Value",
@"Name of undo/redo menu item after Integer text field
was set")];
    }
}

- (IBAction)decimalTextFieldAction:(id)sender {
    if ([[self mySettings] decimalValue] != [sender
floatValue]) {
        [[self mySettings] setDecimalValue:[sender
floatValue]];
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Decimal Value",
@"Name of undo/redo menu item after Decimal text field
was set")];
    }
}

- (IBAction)telephoneTextFieldAction:(id)sender {
    if (![[[self mySettings] telephoneValue]
isEqualToString:[sender stringValue]]) {
        [[self mySettings] setTelephoneValue:[sender
stringValue]];
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Telephone Value",
@"Name of undo/redo menu item after Telephone text field
was set")];
    }
}
```

Note that the `telephoneTextFieldAction:` method must use NSString's `isEqualToString:` method, rather than the direct comparison that is possible with an `int` and a `float`.

You should also change the Speed Limiter text field's action method, in `MyWindowController.m`, so that repeatedly pressing the Tab key on that field will not cause the

same problem.

There is a subtle difference in implementation in the case of the Speed Limiter text field. The text field is secondary to the Speed Limiter slider. The visual state of the Speed Limiter slider is, of course, based on the state of its associated data value in the MySettings model object. When you created the Speed Slider text field, however, you made a point of having it take its value from the slider in order to keep its visual representation synchronized with that of the slider. Although it does not make a difference in behavior here, it seems important to get in the habit of testing a secondary control like the Speed Slider text field against the visual state of the primary control, rather than against the data value in the model object, in order to maintain the secondary nature of the secondary control. In that way, if the GUI ever gets out of sync with the data, you know you should debug the slider and can ignore the secondary text field. Here, therefore, you test the text field's value against the slider's value, not against the MySettings data value.

```
- (IBAction)speedTextFieldAction:(id)sender {
    if ([[self speedSlider] floatValue] != [sender
floatValue]) {
        [[self mySettings] setSpeedValue:[sender
floatValue]];
        [[[self document] undoManager]
setActionName:NSLocalizedString(@"Set Speed Limiter",
@"Name of undo/redo menu item after Speed text field was
set")];
    }
}
```

Note that it can be hard to exercise this textfield to make sure it is working correctly, because the setting of a slider that displays a continuous scale is rounded off in the text field display. If you retype the displayed value into the text field and enter it, the action will usually be registered with the undo manager because you have eliminated fractional values that were previously recorded in the slider and in the data value but were not displayed in the previous setting of the text field. To test this text field, drag the slider all the way to the other end, then retype the value in the text field and enter it; the action will not be registered with the undo manager because there was no fractional part to the setting of the slider.

9. You might wonder whether any action methods other than text field action methods require modification to suppress registration with the undo manager when resetting them to their previous values. Since the application doesn't allow you to tab through controls that are not text fields, it may seem as though the issue is limited to text fields. But it isn't that simple.

   Another way to cause unnecessary registration with the undo manager with respect to some kinds of controls is to simply click on them. Checkboxes do not present this problem, because whenever you click on a checkbox, its value changes and registration with the undo manager is required. Some thought and a little testing reveal, however, that it is possible to click on a control of several other

kinds without changing its value, and in each case the document is dirtied inappropriately. These include clicking on a radio button that is already "on;" clicking on a slider at precisely its current setting; and using a pop-up menu to reselect its current setting. In these cases, the user probably did not intend to make a change to the document, and in any event no change was made. In some cases, the user may simply have been looking to see what other values are available (for example, in the case of a pop-up menu). You should therefore make similar changes to each of the action methods in `MyWindowController.m` shown below. In each case, wrap the existing body of the action method in the indicated test.

In `partyAction:`,

```
if ([[self mySettings] partyValue] != [sender
selectedTag]) {
    …
}
```

Note that you called NSControl's `selectedTag` method to get the current selection of the radio button cluster, instead of the less direct method you used elsewhere in the `partyAction:` method, namely, calling NSControl's `selectedCell` accessor method and then getting its tag. You should go ahead and substitute the simpler method where it appears twice in the `partyAction:` method, just to simplify the code.

In `stateAction:`,

```
if ([[self mySettings] stateValue] != [sender
indexOfSelectedItem]) {
    …
}
```

In `personalityAction:`,

```
if ([[self mySettings] personalityValue] != [sender
floatValue]) {
    …
}
```

In `speedSliderAction:`,

```
if ([[self mySettings] speedValue] != [sender
floatValue]) {
    …
}
```

In `quantumSliderAction:`,

```
if ([[self mySettings] quantumValue] != [sender
floatValue]) {
    …
}
```

In each of `quantumButton1Action:` and `quantumButton2Action:`, it is not the current setting of the sender, a button, but the minimum and maximum acceptable settings of the slider, respectively, that govern the test. Also, these two buttons are secondary controls, so it is the current value of the slider, not the current data value in the MySettings object, that you test. In `quantumButton1Action:`,

```
if ([[self quantumSlider] floatValue] != [[self
quantumSlider] minValue]) {
    …
}
```

And in `quantumButton2Action:`,

```
if ([[self quantumSlider] floatValue] != [[self
quantumSlider] maxValue]) {
    …
}
```

Build and run the application. Turn to the Text Fields pane and press the Tab key to confirm that the insertion point appears in the Integer text field.

Type a digit or three. Then type a character that is not included in the digits "1" through "9", and note that the computer beeps at you and does not display the illegal character. Be sure to try typing an illegal character in several locations, not just at the end of the text field. Also try pasting a string of characters into the field at any location, using both a string of digits and a string that includes one or more nondigits.

Hit the Tab key and note that the insertion point jumps to the Decimal text field. Save the document to disk, then change the value in the Integer text field and choose File > Revert. If the Integer text field reverts to the saved value, you know that hitting the Tab key properly committed the value you first entered.

Hit the tab key several times and see that each text field in turn is made key, and that you can continue tabbing indefinitely to cycle repeatedly through the fields. Watch the window's close button or look at the Edit > Undo menu item after each press of the Tab key to confirm that the action was not registered with the undo manager.

Select the Sliders pane and make sure the Speed Limiter text field is made key when you press the Tab key. Click on the currrent setting of any of the sliders to verify that the action was not registered with the undo manager. Set the Quantum slider to its highest or lowest setting then click the Highest or Lowest button to confirm that this action, too, is not registered with the undo manager.

Select the Buttons pane and reselect the current setting of the Party radio button cluster and the State pop-up menu to confirm that neither action is registered with the undo manager.

You have now learned how to apply a simple character filtering formatter to a text field. In the next Step you will take this knowledge a step forward, applying a slightly more complex custom formatter and adding the ability, not only to filter, but also to format a text field entry on the fly.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

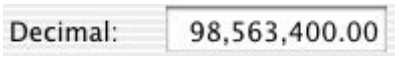
By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 5: User controls—Text fields (formatters)

Step 2: On-the-fly input filtering and formatting for decimal values

- Highlights
  - Obtaining localized user default values from NSUserDefaults
  - Creating and using a custom character set for membership testing
  - Limiting typing in a text field to positive decimal values (filtering for numeric digits plus a localized decimal separator)
  - Creating and using a scanner to remove unwanted characters from a string
  - Formatting a text field on the fly by inserting thousands separators automatically and adjusting the insertion point
  - Recognizing a control character or function key typed on the keyboard



In *Step 1*, you implemented simple on-the-fly input filtering, without any on-the-fly formatting, for the first of three text fields. In this *Step 2*, you will implement a more complex formatter for the second field, which will let the user type positive decimal values using the numeric digits plus a localized decimal point; typing any other character will cause the computer not only to beep but also to present a sheet explaining the input error. You will also provide formatting on the fly by inserting localized thousands separators in the correct locations as the user types, deletes, forward deletes, cuts and pastes individual characters and selections.

You already created the Decimal text field in *Step 1*, so you can get right to work on the custom formatter.

1. First, in `MyWindowController.m`, revise the `control:didFailToValidatePartialString:errorDescription:` delegate method in the Formatter errors section to test whether the Decimal text field was the source of the validation error. If it was, and if you determine that there was really a validation error, you will provide the user with both a beep and a sheet. You will write the code for the sheet shortly, using the techniques you learned in *Recipe 4*.

```
- (void)control:(NSControl *)control didFailToValidatePartialString:(NSString
*)string errorDescription:(NSString *)error {
    if (control == [self integerTextField]) {
        NSBeep();
    } else if (control == [self decimalTextField]) {
        if (error != nil) {
            NSBeep();
            [self sheetForDecimalTextFieldValidationFailure:string
errorDescription:error];
        }
    }
}
```

   Here, as in *Recipe 4, Step 2*, the custom method you are about to write to present the sheet, `sheetForDecimalTextFieldValidationFailure:errorDescription:`, will return `NO` as a result in order to prevent

the invalid string from being displayed. However, the `control:didFailToValidatePartialString:errorDescription:` delegate method does not need to return that result to the caller, as did the `control:didFailToFormatString:errorDescription:` method, so you ignore the result here.

The new section of this method tests the `errorDescription` parameter to determine whether the formatter did not return `nil`. Anticipating some code that you will write at the end of this Step, this reflects a convention you will adopt of returning `nil` instead of an error description string from an on-the-fly formatter to signal that no error has occurred. This is necessary when the formatter substitutes some text that it has edited for the text that the user actually typed, because Cocoa invokes the `control:didFailToValidatePartialString:errorDescription:` method in that situation even though there has not been an error. You don't want to display an alert sheet when there is no error, but Cocoa lets your application know that the user's typing has been edited by your formatter in case you want to do something with that information.

2. To ensure that this delegate method will be called automatically by NSControl, you must appoint MyWindowController as the delegate of this control. In Interface Builder, Control-drag from the Decimal text field to the File's Owner icon, which you recall is a proxy for the MyWindowController object, then connect the `delegate` target.

3. Next, attach the formatter to the control, leaving the code for the sheet and the formatter itself until later in this Step.

    a. When you create the formatter, you will call it MyDecimalFilter, a subclass of NSNumberFormatter. In `MyWindowController.m`, in the new Formatter constructors section after the `makeIntegerFilter` method that you wrote in the previous Step, declare the method that will instantiate a MyDecimalFilter object and attach it to the Decimal text field, as follows:

```
- (void)makeDecimalFilter {
    MyDecimalFilter *decimalFilter = [[MyDecimalFilter alloc] init];
    [decimalFilter setFormat:@"#,##0.00"];
    [[[self decimalTextField] cell] setFormatter:decimalFilter];
}
```

    Declare this method in `MyWindowController.h` after the `makeIntegerFilter` declaration:

```
- (void)makeDecimalFilter;
```

    You must also import `MyDecimalFilter.h` into `MyWindowController.m` by adding the following line to the end of the import directives at the top of the file:

```
#import "MyDecimalFilter.h"
```

    b. In the `makeFormatters` method you added to `MyWindowController.m` in the previous Step, insert the following in order to create the new formatter when the window opens:

```
[self makeDecimalFilter];
```

    c. Add the following line to the `dealloc` method in `MyWindowController.m`, just after the Integer text field's formatter is released, to release this formatter, as well, when the window is closed.

```
[[[self decimalTextField] formatter] release];
```

4. Now create the sheet that will explain to the user what's wrong when an illegal character is typed. This sheet is very simple, requiring only some informative text and an OK button. No callback method is required because the sheet presents the user with no options other than to click the OK button when an illegal character is typed; `nil` is passed in both callback selector parameters to the `NSBeginAlertSheet()` function. The message text for the sheet could have been defined in this method, as you have done with your other sheets, but here you want the message text to be added by the formatter you will shortly create. Instead, therefore, you

simply accept the `error` string value that will be passed in to the `sheetForDecimalTextFieldValidationFailure:errorDescription:` method from the formatter and pass it along to the sheet as the first parameter of the `NSBeginAlertSheet()` function. Add this method at the end of the `MyWindowController.m` file:

```
// Decimal text field

- (BOOL)sheetForDecimalTextFieldValidationFailure:(NSString *)string
errorDescription:(NSString *)error {

    NSString *alertInformation = NSLocalizedString(@"The Decimal text field
accepts any of the digits “0”–“9” and a decimal point.", @"Informative text for
alert posed by Decimal text field when invalid character is typed");
    NSString *defaultButtonString = NSLocalizedString(@"OK", @"Name of OK
button");

    NSBeginAlertSheet(error, defaultButtonString, nil, nil, [self window],
self, nil, nil, NULL, alertInformation);

    return NO; // reject bad string
}
```

Add the declaration at the end of `MyWindowController.h`:

```
// Decimal text field

- (BOOL)sheetForDecimalTextFieldValidationFailure:(NSString *)string
errorDescription:(NSString *)error;
```

Be sure to add the informative text for the sheet to the `Localizable.strings` file. The localized name of the OK button has already been provided. Add the following at the end of the `Localizable.strings` file:

```
/* Decimal text field alert */
/* Informative text for alert posed by Decimal text field when invalid
character is typed */
"The Decimal text field accepts any of the digits “0”–“9” and a decimal point."
= "The Decimal text field accepts any of the digits “0”–“9” and a decimal
point."
```

5. You are now ready to write the new formatter.

   Before you turn to the code, consider what your strategy will be. First, on the broadest view, it should do two things: filter out invalid characters (characters other than the decimal digits and a decimal separator), and insert thousands separators automatically. In addition, it should do these things in a manner that is intuitive and transparent to the user. For example, if the user deletes or cuts something, it should be possible to retype it or paste it back in immediately, without having to use the arrow keys to reposition the insertion point. It begins to dawn on you that accomplishing this may involve some additional work that was not required in the Integer text field formatter that you just completed, such as adjusting the location of the insertion point to accommodate the editing done by the formatter. It seems reasonable to tackle the filtering and the formatting tasks sequentially, as independent sections of code, but it appears that each section will have to keep in mind what the other section does.

   The filter section should be similar to what you wrote in *Step 1*, but there are some complexities to consider that you didn't address there. For one thing, what if the user cuts a portion of this very text field and it includes one or more of the thousands separators that will be added automatically by the formatter, and then tries to paste it back into the field? Although you haven't specified that the user should be allowed to type thousands separators, usability considerations suggest that you will have to write the filter in such a way that it doesn't reject pasted thousands separators. Finally, this is a decimal number formatter, and a decimal separator may therefore be typed or pasted by the user. But what if the user types two of them, or pastes a string that would give the field two or more of them?

The filter will obviously have to count decimal separators and reject any effort to insert more than one of them.

The formatter section is also a can of worms. From the moment the user first types or pastes characters into the text field, you not only have to consider filtering out invalid charcters, but also where to place the thousands separators that are the formatter's responsibility. Furthermore, the second time the user types or pastes into the field, you have to consider getting rid of the thousands separators that were added the first time around, because they are now probably postioned incorrectly. Both deleting and inserting thousands separators will require careful monitoring and adjustment of the insertion point's location. Finally, you may not anticipate it now, but you will soon discover that the user's typing of the Delete key or the Forward Delete key can present subtle problems when positioning the insertion point.

In short, writing a formatter is no easy task. You must take care to write a formatter so that it makes no assumptions about where the insertion point is (start, end or middle of the text field), nor about what was selected (all of the text field, nothing or a substring), nor about whether the user is typing, cutting, deleting or forward deleting, or pasting. To do this, you must evaluate a snapshot of the contents of the text field after each character was typed or each string was pasted. It can be quite difficult to design an algorithm that will work in all possible scenarios, and even more difficult to simplify and generalize it instead of dealing with every issue as a special case. Trial and error is not only inevitable, but necessary to make sure you haven't left a bug that a user is bound to discover.

a. Create new Cocoa header and source files and name them **`MyDecimalFilter.h`** and **`MyDecimalFilter.m`**, respectively. Choose Project > Add Files…, add the two new files, and drag their icons into the Formatters group in the Groups & Files pane of the Project Builder window, if necessary.

b. Set up the header file `MyDecimalFilter.h` so that it imports the Cocoa umbrella framework and inherits from NSNumberFormatter. The skeleton should look like this:

```
#import <Cocoa/Cocoa.h>

@interface MyDecimalFilter : NSNumberFormatter {
}

@end
```

Add the following instance variable declarations, which you will use shortly, inside the curly braces of the interface declaration in `MyDecimalFilter.h`:

```
@protected
NSString *localizedDecimalSeparatorString;
NSString *localizedThousandsSeparatorString;
NSCharacterSet *invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet;
```

You can probably deduce from the instance variables how the formatter is going to handle its filtering function. It will first learn which character is used in the machine's locale as the decimal separator and which as the thousands separator, and set up a string version of each of them for use with the Cocoa methods you will use in the formatter. Then it will create a custom character set, which you call `invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet`, consisting of every character *except* the ten numerical digits and these localized decimal and thousands separator characters. Each character that the user types into the text field will be tested for membership in this custom character set and rejected if it falls within the set. Because of this logic, as you learned in *Step 1*, it is convenient to use an inverted set instead of the set of decimal digits and separators. In that Step, you did not have to create a custom character set to be inverted, because Cocoa comes equipped with a pre-built character set for the numeric digits; here, you will first create a custom character set, then invert it.

You may be surprised to see that the filter will allow the user to type and paste thousands separators. This turns out to be the easiest way to solve the problem, mentioned above, of letting the user cut a portion of the field that contains thousands delimiters and paste it right back into the field. You simply make thousands separators legal, and reposition them as necessary in the formatting block, should the user choose to type or paste one of them. The user's experience will be that typing a thousands separator accomplishes nothing, because the thousands separator will instantly disappear or be repositioned by the formatter.

The character set is created when the formatter is initialized, in order to minimize the amount of processing required while the

user is typing. Machines are fast enough these days that this precaution may seem antediluvian, but it is a good idea to minimize the processing load during typing, if for no other reason than to reserve as much time as possible for background tasks. You should read the NSCharacterSet reference document to see what steps that Apple recommends you take to avoid slowdowns that can occur if you misuse character sets. Here, the formatter will be created when a new window is opened, which is an occasion when a little time is available to take care of things like setting up a character set. You will also create the inverted character set when the formatter is initialized; although the inversion process was apparently fast enough in the Integer filter that it could safely be performed every time the user typed a character, you might as well take care of it once in the `init` method of the Decimal filter.

Notice that you have not declared any accessor methods for the instance variables. The formatter will be called upon to validate every character as the user types it, and another way that you can avoid using up processor time unnecessarily is to dispense with accessor methods and call the instance variables directly while the user is typing. You used the `@protected` compiler directive to allow any subclass inheriting from this formatter to access the same instance variables despite the absence of an accessor method.

c. Set up the source file `MyDecimalFilter.m` so that it imports its header file, and provide the implementation skeleton, as follows:

```
#import "MyDecimalFilter.h"

@implementation MyDecimalFilter

@end
```

d. Now, add the `init` and `dealloc` methods to `MyDecimalFilter.m`:

```
//Initialization

- (id)init {
    if (self = [super init]) {
        NSMutableCharacterSet *tempSet;
        NSCharacterSet *decimalDigitPlusLocalizedSeparatorsCharacterSet;

        localizedDecimalSeparatorString = [[NSUserDefaults
standardUserDefaults] objectForKey:NSDecimalSeparator];
        localizedThousandsSeparatorString = [[NSUserDefaults
standardUserDefaults] objectForKey:NSThousandsSeparator];

        tempSet = [[NSCharacterSet decimalDigitCharacterSet] mutableCopy];
        [tempSet addCharactersInString:[localizedDecimalSeparatorString
stringByAppendingString:localizedThousandsSeparatorString]];
        decimalDigitPlusLocalizedSeparatorsCharacterSet = [tempSet copy];
        [tempSet release];

        invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet =
[[decimalDigitPlusLocalizedSeparatorsCharacterSet invertedSet] retain];
        [decimalDigitPlusLocalizedSeparatorsCharacterSet release];
    }
    return self;
}

- (void)dealloc {
    [invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet release];
    [super dealloc];
}
```

As override methods, these do not require declaration.

First, you set up two of the instance variables to refer to the localized decimal and thousands separators. The NSUserDefaults class reference document contains information about important localized user defaults that are built into Cocoa, which you can obtain using the provided constants, such as NSDecimalSeparator and NSThousandsSeparator, as keys to determine what values are being used on the computer based on its current locale. The values you obtain for the decimal and thousands separators are strings, which is just what you need for the Cocoa methods you will use here.

Second, you create the custom inverted character set. You do this in two steps, one to create an intermediate immutable character set containing the decimal digits and both separators, and a second to create the inverse of that set. The logic of the filter you are about to write requires the inverted set.

The technique used to create the first character set comes straight out of the NSCharacterSet documentation. You first create a mutable copy of the built-in Cocoa `decimalDigitCharacterSet` character set, which you used in the previous Step, using NSObject's `mutableCopy` method, and assign it to a temporary local variable whose declared type is `NSMutableCharacterSet`. Next, you add the localized decimal and thousands separator strings to it using an NSCharacterSet method designed for this purpose, `addCharactersInString:`. You then copy the temporary mutable character set into another, intermediate local variable, `decimalDigitPlusLocalizedSeparatorsCharacterSet`, this one declared as an immutable character set. Finally, you release the temporary mutable object. By using NSObject's `copy` method to assign the new character set to the intermediate local variable, you ensure that it holds an immutable character set. You want the intermediate character set to be immutable because, according to the NSCharacterSet reference document, it is much faster to invert an immutable character set than a mutable character set.

Creating the second character set is a simple matter of calling NSCharacterSet's `invertedSet` method on the intermediate character set, retaining the result and assigning it to the instance variable that you have already declared in the interface file as an immutable character set, `invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet`, and releasing the intermediate local variable. You make the instance variable immutable because the documentation tells you that an immutable character set is much faster when used in a membership test than the temporary mutable character set that you had to create in order to add the separators to it.

The `mutableCopy` method that assigned the initial, temporary object to the temporary variable, like the allocation of a new object, implicitly retained it, thus requiring you to release it, which you do here before execution leaves the `init` method. Similarly, the `copy` method used to assign a value to the intermediate variable also performed an implicit `retain` and therefore requires a matching `release`. Finally, you explicitly retained the instance variable, in order to ensure that it will remain in memory for use when the user types into the text field to which the formatter is attached. The character set will last for the life of the formatter, and you have written the application to keep the formatter alive for the life of the window. Thus, the `dealloc` method must release the new character set when the formatter is deallocated as the window is closed.

e. Start writing the input filtering and formatting method. This method is complex, and you will take it one step at a time. First, add the signature and the first statement in the method to `MyDecimalFilter.m`:

```
// Input filter and formatter

- (BOOL)isPartialStringValid:(NSString **)partialStringPtr
proposedSelectedRange:(NSRangePointer)proposedSelRangePtr
originalString:(NSString *)origString
originalSelectedRange:(NSRange)origSelRange errorDescription:(NSString
**)error;

     NSString *lastInsertedString = [*partialStringPtr
substringWithRange:NSMakeRange(origSelRange.location, proposedSelRangePtr-
>location - origSelRange.location)];
}
```

This first statement obtains and places into the `lastInsertedString` local variable what the user typed or pasted into the field, in the form of a string. You already used this technique in *Step 1*, where it was buried in a complicated `if` test. Here, you have pulled it out and used a local variable to hold the user's input.

Note that the `location` element of the proposed selection range is accessed here using the optional C structure pointer operator, in `proposedSelRangePtr->location`, instead of the structure member operator that you used in *Step 1*, `(*proposedSelRangePtr).location`. Either form is acceptable.

f. The next step is to filter the `lastInsertedString` variable for invalid characters to be rejected. Add the first part of the filter section, immediately following the statement you added in the previous instruction:

```
// Filter out invalid printable characters:

    if ([lastInsertedString
rangeOfCharacterFromSet:invertedDecimalDigitPlusLocalizedSeparatorsCharacterSet
options:NSLiteralSearch].location != NSNotFound) {
        *error = [NSString stringWithFormat:NSLocalizedString(@"“%@” is not
valid here", @"Presented when typed or pasted value contains a character other
than a numeric digit or decimal separator"), lastInsertedString];
        return NO;
    }
```

This code is self explanatory, given your experience with this technique in *Step 1*. One of the very useful `rangeOfCharacter...` methods from NSString is used to determine whether the `lastInsertedString` variable contains any character from the inverted set you created earlier. If so, it must be rejected with an error message and a return value of `NO`.

g. The `errorDescription` parameter value should, of course, be localized. Add the following to the new Formatters section of the `Localizable.strings` file (this message isn't very smooth, but it will do for your purposes):

```
/* Presented when typed or pasted value contains a character other than a
numeric digit or decimal separator */
"“%c” is not valid here" = "“%c” is not valid here"
```

h. This first part of the formatter's main method is only the beginning. It is now properly filtering out all characters except the digits and the localized separators. However, it allows the user to type an unlimited number of decimal separator characters, which is, of course, inappropriate for a field that is to contain a valid decimal number. In addition, you have not yet fulfilled the promise to provide localized thousands separators on the fly.

A reasonable approach to ensure that the field contains only a single decimal separator is to test the entire contents of the text field after the input string was inserted to see whether it contains the decimal separator and, if it does, to search the remainder of the field to see whether it contains a second decimal separator.

Complete the section of the method you just created in `MyDecimalFilter.m` by adding the following `else` clause after the existing `if` clause:

```
} else {
    decimalSeparatorLocation = [*partialStringPtr
rangeOfString:localizedDecimalSeparatorString
options:NSLiteralSearch].location;
    if ((decimalSeparatorLocation != NSNotFound) && ([*partialStringPtr
length] - 1 > decimalSeparatorLocation) && ([[*partialStringPtr
substringFromIndex:decimalSeparatorLocation + 1]
rangeOfString:localizedDecimalSeparatorString
options:NSLiteralSearch].location != NSNotFound)) {
        *error = [NSString stringWithFormat:NSLocalizedString(@"“%@” can't
appear more than once here", @"Presented when typed or pasted value
contains a second decimal separator"), localizedDecimalSeparatorString];
        return NO;
```

```
    }
}
```

This clause analyzes the entire contents of the text field in `*partialStringPtr`, rather than analyzing only the most recent input string, because the text field may already contain a decimal separator that was added earlier, when it was legal to do so. Two or more decimal separators anywhere in the field would be illegal, however, so you start by checking whether there is any decimal separator in the field. Evaluation of the string is handled by two calls to NSString's `rangeOfString:options:` method, which you first saw in *Step 1*.

Here, three tests are made in the `if` clause. First, if a decimal separator is found nowhere in the text field, the flow of execution leaves the `if` clause. Otherwise, if a decimal separator is found, then its location in the field is tested; if it is at the end of the field, execution leaves the `if` clause because by definition there can't be a subsequent decimal separator in the field. Otherwise, finally, the portion of the field remaining following the first decimal separator is extracted using NSString's `substringFromIndex:` method, and the `rangeOfString:options:` method is used again to detect a second decimal separator, if any. If one is found, an error message is generated and the method returns `NO`.

i. The `errorDescription` parameter value should, again, be localized. Add the following to the new Formatters section of the `Localizable.strings` file (this message isn't very smooth, but it will do for your purposes):

```
/* Presented when typed or pasted value contains a second decimal
separator */
"“%c” can't appear more than once here" = "“%c” can't appear more than
once here"
```

j. What remains is to fulfill the promise to provide automatic, on-the-fly localized thousands separators as the user types. The technique used will be similar in concept to that used above to filter the text field. You have already defined the thousands separator instance variable. Using it, your strategy will be to strip any existing thousands separators from the contents of the text field, then insert new thousands separators wherever needed, adjusting the insertion point backwards and forwards at every stage to keep it synchronized with the contents of the field.

First, add the following code to the formatter's main method, following the `if` and `else` clauses that you wrote above:

```
// Adjust location of thousands separators:

scanner = [NSScanner localizedScannerWithString:*partialStringPtr];
*partialStringPtr = @"";
while (![scanner isAtEnd]) {
    NSString *tempString;
    if ([scanner scanUpToString:localizedThousandsSeparatorString
intoString:&tempString]) {
        *partialStringPtr = [*partialStringPtr
stringByAppendingString:tempString];
    } else if ([scanner scanString:localizedThousandsSeparatorString
intoString:nil]) {
        if ([scanner scanLocation] <= (origSelRange.location +
[lastInsertedString length])) {
            proposedSelRangePtr->location--;
        }
    }
}
```

You also have to declare the `scanner` variable, at the top of the formatter's main method, as follows:

```
NSScanner *scanner;
```

This is your first encounter with a scanner, a fascinating and useful Cocoa object. A scanner may not be appropriate in every situation where you might consider using one, because more efficient special-purpose methods may be available, but a scanner can be very useful indeed for some purposes. Scanners are useful because of their generality; they make it easy to do something like stripping thousands separators from a string, without requiring you to dive very deeply into management of indices. In this context, this scanner is plenty quick enough to let a fast typist work the keyboard without hindrance. Be sure to read the NSScanner reference document in conjunction with this code.

What this section does is, first, to create a scanner object using one of the provided `With`... methods, in this case, with the string that forms the current contents of the text field. Once the string is safely copied into the scanner, the code reinitializes the string as it exists in the text field to an empty string, preparatory to concatenating numbers into it from the copy of the original string now residing in the scanner. It then enters a `while` loop, which will be terminated when the NSScanner `isAtEnd` method returns `YES`. In the `while` loop, the scanner successively scans a run of valid digits or decimal separators into a temporary string variable and concatenates the temporary variable into the reinitialized contents of the text field, then scans a thousands separator, if one is found, into oblivion. Whenever a thousands separator is found and discarded, the insertion point is decremented if the thousands separator was located to the left of the insertion point, in order to move the insertion point to the left in sync with the left shift of all characters that were located to the right of the discarded thousands separator. (The insertion point will be incremented again when thousands separators are added back to the field in the next block of code.) The temporary string variable is concatenated into the text field on each pass until the text field is fully reconstituted, minus the thousands separators, and the insertion point remains adjacent to the digit to which it was initially adjacent.

The conditions for decrementing the insertion point are very sensitive. The scanner's `scanLocation` is used to determine how much progress has been made. Since the string within the scanner is not altered as the scanner operates, the values of `scanLocation` are measured in every iteration from the original beginning of the text field. For this reason, these values are measured against the starting insertion point, which is calculated as the sum of the original insertion point before the user typed or pasted, `origSelRange.location`, and the length of the `lastInsertedString` string. It isn't appropriate to use the constantly changing adjusted selection point, `(*proposedSelrangePtr).location`. The "less than or equal to" inequality operator was chosen based on visualization of what is happening to the string dynamically as the code executes.

k. Next, you must add thousands separators back into the text field in appropriate locations.

A constraint you must observe is to work only with the integer part of the decimal number in the text field, which you do by noting the location of the decimal separator, if any.

One way to position thousands separators appropriately is to apply a log function to the numerical value of the string. Here, however, you have already stripped out the old thousands separators, so you can position the new separators instead by counting characters by three's in the integer part of the decimal number. You will do this by using the standard C modulus operator, %. The hardest part of this is to get the location of the insertion point right after inserting the thousands separators.

Add the following section after the section you just completed:

```
thousandsSeparatorLocation = 1;
decimalSeparatorLocation = [*partialStringPtr
rangeOfString:localizedDecimalSeparatorString
options:NSLiteralSearch].location;
remainingIntegerLength = (decimalSeparatorLocation == NSNotFound) ?
[*partialStringPtr length] - 1 : decimalSeparatorLocation - 1;
while (remainingIntegerLength > 0) {
    if (remainingIntegerLength % 3 == 0) {
        *partialStringPtr = [NSString stringWithFormat:@"%@%@%@",
[*partialStringPtr substringToIndex:thousandsSeparatorLocation],
localizedThousandsSeparatorString, [*partialStringPtr
substringFromIndex:thousandsSeparatorLocation]];
        if ((thousandsSeparatorLocation <= proposedSelRangePtr->location)
&& !((lastTypedChar == NSDeleteCharacter) && [[*partialStringPtr
substringWithRange:NSMakeRange(proposedSelRangePtr->location, 1)]
isEqualToString:localizedThousandsSeparatorString])) {
            proposedSelRangePtr->location++;
```

```
        }
        thousandsSeparatorLocation++;
    }
    thousandsSeparatorLocation++;
    remainingIntegerLength--;
}
```

This block first defines an intermediate variable and two working local variables; one of the latter, `thousandsSeparatorLocation`, tracks the location where the next thousands separator is to be inserted, and one, `remainingIntegerLength`, tracks how many characters following that remain to be processed in the integer part of the string. The C conditional expression `?` is used to concisely set the `remainingIntegerLength` variable depending on whether a decimal separator is present in the string. The first thousands separator will be inserted at location 1 when a total of 4 characters have been processed, with 3 characters following the first thousands separator location. In a `while` loop that is terminated when the remaining length is whittled down to zero, a thousands separator is added in every iteration in which the remaining length modulo 3 is 0. In all iterations, the two working local variables are incremented and decremented, respectively, in preparation for the next iteration. When an iteration is reached that requires insertion of a thousands separator, the NSString class method `stringWithFormat:` is called to concatenate the two substrings on either side of the thousands separator location, with a localized thousands separator sandwiched between them. In addition, the `thousandsSeparatorLocation` is incremented an extra notch.

The insertion point is incremented every time a thousands separator is inserted to the left of the currently proposed insertion point, in order to move the insertion point to the right in sync with the right shift of the remaining characters.

l. One insertion point consideration is devlishly hard to resolve: the effect of the user's pressing the Delete key or the Forward Delete key.

Part of the difficulty lies in the fact that the original selection range's `location` and `length` elements do not behave as one would expect when either of these keys is pressed. Cocoa acts as if it has already carried out the actions of the Delete and Forward Delete keys before invoking the `isPartialStringValid:...` method, when reporting `origSelRange`. The original selection range's `location` is reported as if it were one less than it actually was before the delete key was pressed, and it is reported as unchanged after the forward delete key was pressed. In both cases, the original selection range's `length` is reported as `1` even if there was no selection before the delete or forward delete key was pressed. Despite these surprises, the original selection string still includes the character that was deleted, as expected.

You deal with the Delete and Forward Delete issue here by ignoring the original string and original selection range parameters and relying only on the *proposed* selection range location and length, which are reported as you would expect. The effect of the Forward Delete key is accommodated in the basic code, since the `location` element doesn't change then. The action of the Delete key is special-cased. If the Delete key is detected and the character at the proposed insertion point is a thousands separator, the proposed insertion point is not incremented even if it would have been otherwise. In both cases, the effect is to skip over a thousands separator without deleting it whenever the Delete or Forward Delete key is pressed. This is the desired behavior, because each keypress in this on-the-fly formatter is supposed to leave thousands separators in the correct locations. Although the user is allowed to type or delete a thousands separator, the appearance of the field is controlled by the formatter and the user only sees thousands separators where they belong.

The code in instruction 5.k., above, used four variables that have not yet been declared, and this snippet of code: `(lastTypedChar == NSDeleteCharacter)` to detect the user's pressing of the Delete key. You now need to declare these variables and provide the code to make the Delete key snippet work. At the top of the formatter's main method, after the declaration of `scanner`, add the following:

```
int thousandsSeparatorLocation;
int decimalSeparatorLocation;
int remainingIntegerLength;

unichar lastTypedChar = [[[NSApp currentEvent] characters]
characterAtIndex:0];
```

The character last typed by the user is obtained from the most recent system event that was generated by the user's typing activity. You can't use the last printable character, as it appears in the `*partialStringPtr` parameter, because your purpose is to detect a control character, which isn't printable and so doesn't appear in the text field. The NSApplication object provides the `currentEvent` method, from which you can always determine the user's most recent key down, key up, or mouse actions, along with many others, even if they aren't printable. `NSApp` is a global variable provided by Cocoa to make it easy for you to access your application object for information like this. (You could also obtain the current event from the window, but in this situation the application's current event is just as good.)

You should read the NSEvent reference document at this point to see how much information the NSEvent class makes available to you when you need to know what the user is doing with the keyboard or other input devices. You should also read the NSResponder and NSTextView reference documents, which also have useful information about handling user input. One of NSEvent's methods is `characters`, a string that contains the Unicode character or characters that were generated by the user's last keypress, including nonprintable characters. Although some keypresses can generate more than one Unicode character, you are only concerned about simple characters here and can therefore make do with the character at index `0` of the string.

Where did the constant for the Delete key come from? As with much of Cocoa programming, you just have to become familiar with the AppKit and Foundation classes to learn where useful things like this are hidden. It turns out that the constant for the Delete character (known to older typists as Backspace), along with several other traditional typewriter control characters such as the carriage return, is defined in `NSText.h`. All of the control character constants there, including the `NSDeleteCharacter` constant used here, end with the word "Character." If you needed it, the constant for the Forward Delete key is defined in `NSEvent.h`, along with many computer-oriented function keys, including all of the function keys F1, F2, etc. They all end with the words "FunctionKey," including the `NSDeleteFunctionKey` constant you could use to detect the Forward Delete key.

m. Finally, the method must return some values. Insert the following code at the end of the method:

```
error = nil;
return NO;
```

The method result is returned as `NO`. As the NSFormatter reference document describes, this result, in conjunction with a new string value passed by reference in the `partialStringPtr` parameter, tells Cocoa to display the edited string. This result also triggers the `control:didFailToValidatePartialString:errorDescription:` method, however, and, without something more, it would sound a beep and trigger an alert sheet in the Vermont Recipes application. As described near the beginning of this Step, you therefore adopt the convention of returning `nil` in the `errorDescription` parameter, to signal that no error has occurred and nothing should be done to disturb the peace.

You took care of connecting the Decimal text field in the previous Step, so you can build and run the application now. Select the Text Fields pane and press the Tab key twice to move the insertion point to the Decimal text field, or click directly in it.

Type a digit or three. Then type a character that is not included in the digits "1" through "9" and is not the localized decimal character. Note that the computer beeps at you and presents a sheet, and it does not display the illegal character.

Type some more digits, and watch as localized thousands separators appear in the proper locations automatically. Use the Delete and Forward Delete keys and watch as the thousands separators continue to be positioned correctly. Be sure to try selecting various portions of the number and deleting, forward deleting, typing, cutting and pasting. Pasting a string containing illegal characters generates an error message. Note also that traditional means of selecting text work as they are supposed to, such as holding down the Shift key while using the arrow keys. Verify that all works as you would expect, and that the insertion point always ends up at a point that allows you to continue typing or to resume typing where a deletion occurred. Try typing or pasting a thousands separator explicitly, and you will find that, although no error is generated, the thousands separators appear only where they are supposed to, not necessarily where you tried to put them.

It is particularly interesting to place the insertion point immediately before or after a thousands separator, then alternately hit the Delete and Forward Delete keys. The insertion point moves back and forth to either side of the thousands separator, without deleting it or in any way altering the correct display of thousands separators.

Try typing a decimal separator and additional digits to form a fractional part of the number. Then try all of the above tests again to see

whether you are happy with the way on-the-fly formatting works in that case. If so, isn't it remarkable that you achieved this without giving it any significant thought?

Finally, hit the Enter key (or click on another text field or Tab out of the Decimal field) while a number with several digits in the fractional part is in the field. Notice that the fractional part is reduced to two digits, rounding the second appropriately. If you save the file and examine it in PropertyListEditor, you will see that the full precision you typed is preserved; the rounding occurs only in the display.

The truncation of the displayed precision occurred because, back in instruction 3.a., above, you called NSNumberFormatter's `setFormat:` method with a template string specifying display of two-digit decimal precision. As you see, the `setFormat:` method provides formatting that is applied only when editing in the text field has ended. Depending on your application's design, you might prefer not to invoke any formatting at that point, but instead simply to accept the text field as the user's typing and the formatter's on-the-fly formatting have left it. The choice is yours.

But what do you do when you want to write a formatter that can't be handled by NSNumberFormatter's (or NSDateFormatter's) built-in templates? The NSFormatter reference document challenge's the reader to write a custom formatter for telephone numbers, part numbers, or Social Security numbers. You will take up the telephone number challenge in the next Step, *Step 3*. There, you will learn how to write a complete formatter, one that handles both on-the-fly and end-of-editing formatting. You will discover that, instead of overriding the one method that you overrode in this and the previous Step, you must subclass NSFormatter itself and override three of its methods.

Vermont Recipes—A Cocoa Cookbook for Mac OS X

By Bill Cheeseman

August 9, 2001 - 6:00 AM

Recipe 5: User controls—Text fields (formatters)

Step 3: A complete custom formatter for conventional North American telephone numbers

- Highlights
  - Writing a custom formatter by subclassing NSFormatter
  - Giving a formatter access to the user control to which it is attached

In *Step 2*, you implemented an on-the-fly decimal number filter and formatter that inserts localized thousands separators in the correct positions as the user types, deletes, forward deletes, cuts and pastes individual characters and selections. In this *Step 3*, you will use similar techniques to implement a telephone number filter and formatter, enabling users to type a conventional North American telephone number in the form "(800) 555-1212" without having to type any of the separator characters.

One of the reasons you will tackle a telephone number formatter is that Apple's NSFormatter reference document more or less challenges you to write one. In addition, however, this formatter gives you an opportunity to apply what you learned in *Step 2* to a more difficult problem, one that requires more general techniques than those you used in *Step 2*. Using the new techniques you will learn here, you should be able to write virtually any formatter.

A telephone number formatter presents these new issues, among others:

- There are several different separator characters—namely, the open and close parentheses, the space character and the dash—instead of just the one, a localized thousands separator. It will not be possible to isolate a single string in your code, this time around; you must instead isolate a set of several characters.
- Two of the separators, the close parenthesis and the space character, always appear together. You will no longer be able to deal with insertion point position adjustments in increments of a single character, but will have to take into account the length of a separator string containing multiple characters.
- A telephone number has a fixed, mandatory length: fourteen characters, including the separators. You must now deal appropriately with the user's attempts to enter strings that are too short or too long.

In addition, you will subclass NSFormatter, rather than NSNumberFormatter, since a telephone number does not need the various numerical formatting routines that are provided in NSNumberFormatter. Whenever you subclass NSFormatter directly, you must override at least a couple of additional methods.

Despite these new issues, you will be surprised to discover that the code for the on-the-fly portion of the telephone number formatter is just as short as that for the decimal number formatter. This is mostly because Cocoa's built-in routines for dealing with character sets are just about as simple as those that deal with strings. It is very easy to subsitute a character set method for a string method when you must deal with a disparate set of multiple separator characters.

Because the basic outline of the telephone number formatter is already familiar to you, we will provide the repetitious code here with very little comment, in the same order as it was presented in *Step 2*. The new techniques will, as usual, be highlighted and explained.

You already created the Telephone text field in *Step 1*, so you can get right to work on the custom formatter.

1. In `MyWindowController.m`, revise the `control:didFailToValidatePartialString:errorDescription:` delegate method again to test whether the validation error originated in the Telephone text field. If it did, and if there was really a validation error, you will provide the user with both a beep and a sheet. You will write the code for the sheet shortly.

```
- (void)control:(NSControl *)control didFailToValidatePartialString:(NSString
*)string errorDescription:(NSString *)error {
    if (control == [self integerTextField]) {
        NSBeep();
    } else if (control == [self decimalTextField]) {
        if (error != nil) {
            NSBeep();
            [self sheetForDecimalTextFieldValidationFailure:string
errorDescription:error];
        }
    } else if (control == [self telephoneTextField]) {
        if (error != nil) {
            NSBeep();
            [self sheetForTelephoneTextFieldValidationFailure:string
errorDescription:error];
        }
    }
}
```

2. To ensure that this delegate method will be called automatically by NSControl, you must appoint MyWindowController as the delegate of this control. In Interface Builder, Control-drag from the Telephone text field to the File's Owner icon, then connect the `delegate` target.

3. Attach the formatter to the control, leaving the code for the sheet and the formatter itself until later.

    a. When you create the formatter, you will call it MyTelephoneFormatter, a subclass of NSFormatter. In `MyWindowController.m`, in the new Formatter constructors section after the `makeDecimalFilter` method, declare the method that will instantiate a MyTelephoneFormatter object and attach it to the Telephone text field, as follows:

```
- (void)makeTelephoneFormatter:(NSControl *)control {
    MyTelephoneFormatter *telephoneFormatter = [[MyTelephoneFormatter alloc]
initWithControl:control];
    [[control cell] setFormatter:telephoneFormatter];
}
```

    There is a significant difference between this method and its counterparts from *Steps 1* and *2*. This time, when you create an instance of the formatter, you pass to it a reference to the user control to which it will be attached. This will allow the formatter to capture the user's keystrokes from the control's window, rather than from the application itself as you did in *Step 2*. It also gives the formatter access to the window's field editor for direct manipulation of its contents, which you will use here to deal with the NSFormatter issue described in *Step 1*, and it would allow your formatter to take its cues from the current state of other controls in the same window. You will shortly write the formatter's designated initializer, `initWithControl:`, which is needed to pass the control to the formatter.

    There is a cost associated with passing the user control to the formatter. Unless you take more elaborate steps to control when and how a telephone formatter is attached to a particular text field, you will have to instantiate a new formatter for each individual text field that holds a formatted telephone number. In MyDecimalFilter, you didn't need a reference to the control because you obtained the user's last keypress from `NSApp`, instead of from the control's window. Since `NSApp` is a perfectly good source for this reference, you really only need to pass the control to a formatter if you need it for some other purpose. We illustrate the technique here in case you want to write a more advanced formatter, but in the real world you would be better off not passing in the user control in the telephone formatter, where you can do without it. In fact, it might generally be best to use the technique introduced in *Step 1*, where you used the key window's field editor.

    Declare this method in `MyWindowController.h` after the `makeDecimalFilter` declaration:

```
- (void)makeTelephoneFormatter:(NSControl *)control;
```

You must also import `MyTelephoneFormatter.h` into `MyWindowController.m` by adding the following line to the end of the import directives at the top of the file:

```
#import "MyTelephoneFormatter.h"
```

b. In the `makeFormatters` method that you added to `MyWindowController.m` in *Step 1*, insert the following in order to create the new formatter when the window opens:

```
[self makeTelephoneFormatter: [self telephoneTextField]];
```

c. Add the following line to the `dealloc` method in `MyWindowController.m`, just after the Decimal text field's formatter is released, to release this formatter when the window is closed.

```
[[[self telephoneTextField] formatter] release];
```

4. Now create the sheet that will explain to the user what happened when an illegal character is typed. This sheet is identical to that created in *Step 2* except that the informative text differs. Add this method at the end of the `MyWindowController.m` file:

```
// Telephone text field

- (BOOL)sheetForTelephoneTextFieldValidationFailure:(NSString *)string
errorDescription:(NSString *)error {
    NSString *alertInformation = NSLocalizedString(@"The Telephone field accepts
ten digits “0”–“9” for a telephone number of the form “(800) 555-1212”.",
@"Informative text for alert posed by Telephone text field when invalid character
is typed");
    NSString *defaultButtonString = NSLocalizedString(@"OK", @"Name of OK
button");

    NSBeginAlertSheet(error, defaultButtonString, nil, nil, [self window], self,
nil, nil, NULL, alertInformation);

    return NO;
}
```

Add the declaration at the end of `MyWindowController.h`:

```
// Telephone text field

- (BOOL) sheetForTelephoneTextFieldValidationFailure:(NSString *)string
errorDescription:(NSString *)error;
```

Add the informative text for the sheet to the `Localizable.strings` file. The localized name of the OK button has already been provided. Add the following at the end of the `Localizable.strings` file:

```
/* Telephone text field alert */
/* Informative text for alert posed by Telephone text field when invalid
character is typed */
"The Telephone field accepts ten digits “0”–“9” for a telephone number of the
form “(800) 555-1212”." = "The Telephone field accepts ten digits “0”–“9” for a
telephone number of the form “(800) 555-1212”."
```

5. You are now ready to write the new formatter. Your strategy will be essentially identical to that applied in *Step 2*.

    a. Create new Cocoa header and source files and name them **`MyTelephoneFormatter.h`** and **`MyTelephoneFormatter.m`**, respectively. Choose Project > Add Files…, add the two new files, and drag their icons into the Formatters group in the Groups & Files pane of the Project Builder window, if necessary.

    b. Set up the header file `MyTelephoneFormatter.h` so that it imports the Cocoa umbrella framework and inherits from NSFormatter (*not* NSNumberFormatter this time). The skeleton should look like this:

```
#import <Cocoa/Cocoa.h>

@interface MyTelephoneFormatter : NSFormatter {
}

@end
```

    Add the following instance variable declarations, which you will use shortly, inside the curly braces of the interface declaration in `MyTelephoneFormatter.h`:

```
@protected
NSCharacterSet *telephoneSeparatorsCharacterSet;
NSCharacterSet *invertedDecimalDigitPlusTelephoneSeparatorsCharacterSet;
NSControl *myControl;
```

    Then add a declaration for the designated initializer method to the same file, as set forth below:

```
// Initialization

- (id)initWithControl:(NSControl *)control; // Designated initializer
```

    c. Set up the source file `MyTelephoneFormatter.m` so that it imports its header file, and provide the implementation skeleton, as follows:

```
#import "MyTelephoneFormatter.h"

@implementation MyTelephoneFormatter

@end
```

    d. Now, add the `init`, `initWithControl:` and `dealloc` methods to `MyTelephoneFormatter.m`:

```
//Initialization

- (id)init {
    return [self initWithControl:nil];
}

- (id)initWithControl:(NSControl *)control { // Designated initializer
    if (self = [super init]) {
        NSMutableCharacterSet *tempSet;
        NSCharacterSet *decimalDigitPlusTelephoneSeparatorsCharacterSet;

        telephoneSeparatorsCharacterSet = [[NSCharacterSet
characterSetWithCharactersInString:@"() -"] retain];

        tempSet = [[NSCharacterSet decimalDigitCharacterSet] mutableCopy];
        [tempSet addCharactersInString:@"() -"];
```

```
        decimalDigitPlusTelephoneSeparatorsCharacterSet = [tempSet copy];
        [tempSet release];

        invertedDecimalDigitPlusTelephoneSeparatorsCharacterSet =
[[decimalDigitPlusTelephoneSeparatorsCharacterSet invertedSet] retain];
        [decimalDigitPlusTelephoneSeparatorsCharacterSet release];

        myControl = control;
    }
    return self;
}

- (void)dealloc {
    [telephoneSeparatorsCharacterSet release];
    [invertedDecimalDigitPlusTelephoneSeparatorsCharacterSet release];
    [super dealloc];
}
```

The `init` and `dealloc` methods are straightforward. The designated initializer, `initWithControl:`, requires only a little explanation. It initializes the two character sets that are used via instance variables in this formatter, `telephoneSeparatorsCharacterSet` and `invertedDecimalDigitPlusTelephoneSeparatorsCharacterSet`, retaining both, using techniques you learned in *Step 2*. The `myControl` instance variable is initialized using the reference to the control that is passed into the designated initializer when the formatter is instantiated by the `makeTelephoneFormatterForTelephoneTextField:` method that you wrote a moment ago.

e. Two new methods are required now, in addition to the designated initializer, that weren't required in the decimal number formatter. Both of them, `stringForObjectValue:` and `getObjectValue:forString:errorDescription:`, must always be overridden when you subclass NSFormatter. You didn't have to override them for your previous formatters, because both of those were subclasses of NSNumberFormatter, which already overrides these two methods for you. (The NSFormatter reference document states that a third, similar method, `attributedStringForObjectValue:withDefaultAttributes:`, must also be overridden, but in fact if your text field does not provide text attributes, you need not override it. The default version returns `nil` to inform Cocoa that text attributes are not implemented in the field.)

   Reread the NSFormatter reference document now to see what these two methods do. Basically, one verifies that the object is of the correct type before returning it as a string for formatted display in the text field, and the other converts the text field's string representation of the object to its native object value. The latter is called when the user actually commits the text field's contents, and it can be used to generate an alert sheet if something is wrong with the string's format as finally committed by the user (as you will soon see, there are some formatting tests that can't be performed until the user stops editing the field). In this formatter, both methods are quite simple because the native format of a telephone number is already a string.

   Write these two override methods now. They need not be declared in the header file. In `MyTelephoneFormatter.m`, add the following:

```
// Output validation

- (NSString *)stringForObjectValue:(id)object {
    if (![object isKindOfClass:[NSString class]]) {
        return nil;
    }
    return object;
}

- (BOOL)getObjectValue:(id *)object forString:(NSString *)string
errorDescription:(NSString **)error {
    if (([string length] == 0) || ([string length] == 14)) {
        *object = string;
        return YES;
    } else {
```

```
        *error = NSLocalizedString(@"Telephone number is too short",
@"Presented when telephone number is too short");
        return NO;
    }
}
```

The first of these, `stringForObjectValue:`, is taken straight from the NSFormatter reference document. Its only function is to return `nil` if the incoming `object` is not of the correct class for the field; otherwise, it returns the object as a formatted string. Here, the incoming `object` is the formatted string itself, so no conversion is required. (There is no reason why text fields can't hold objects of other types, instead, as you will see in *Recipe 6*. In the latter case, this method may be a serious workhorse, responsible for all of the formatting done by the formatter when the field is displayed.)

The second, `getObjectValue:forString:errorDescription:`, is somewhat more complex. It takes the incoming string from the text field and returns by reference a corresponding `object` in its native form, indicating success or failure by returning a method result of `YES` or `NO`. Any improper formatting or other error relating to the incoming string can be caught here and turned into an error signal to the window controller's `control:didFailToFormatString:errorDescription:` delegate method when the user commits the data. Here, the incoming string is known to be in proper format, because it has been filtered and formatted on the fly by the main routine of this formatter, which you are about to write.

However, there is one user action that the on-the-fly formatter can't handle, namely, committing a string before its time; that is, entering a string that is too short, forming an incomplete telephone number. The on-the-fly formatter can know when the user has typed too many characters. but it can't know that the user has typed too few characters until the user has pressed the Enter key or otherwise terminated editing of the field. So this job has to be performed by the `getObjectValue:forString:errorDescription:` method. If the string is the correct length, the method returns it by reference as the `object` parameter value and returns `YES` as the function result; if it is too short, it returns an error message by reference and returns `NO`, which will trigger the `control:didFailToFormatString:errorDescription:` delegate method.

The case of a zero-length telephone number is also accepted as valid, in order to permit the user to tab past the field without entering any telephone number. You don't want to trap a user by forcing the entry of a telephone number, since the user might not be able to find the right number to enter until later.

To complete the picture painted by these two methods, now is a good time to update the `control:didFailToFormatString:errorDescription:` delegate method and to provide another alert sheet for the case of a telephone number that is too short. These are self-explanatory. In the Formatter errors section of `MyWindowController.m`, revise the delegate method to read as follows:

```
- (BOOL)control:(NSControl *)control didFailToFormatString:(NSString
*)string errorDescription:(NSString *)error {
    if (control == [self speedTextField]) {
        return [self sheetForSpeedTextFieldFormatFailure:string
errorDescription:error];
    } else if (control == [self telephoneTextField]) {
        NSBeep();
        return [self sheetForTelephoneTextFieldFormatFailure:string
errorDescription:error];
    } else {
        return YES;
    }
}
```

And at the end of `MyWindowController.m`, add this method for the new sheet:

```
- (BOOL)sheetForTelephoneTextFieldFormatFailure:(NSString *)string
errorDescription:(NSString *)error {
    NSString *alertInformation = NSLocalizedString(@"The Telephone text
field requires a complete telephone number of the form “(800) 555-1212”.",
@"Informative text for alert posed by Telephone text field when incomplete
phone number is entered");
    NSString *defaultButtonString = NSLocalizedString(@"OK", @"Name of OK
button");

    NSBeginAlertSheet(error, defaultButtonString, nil, nil, [self window],
self, nil, nil, NULL, alertInformation);

    return NO;
}
```

Finally, add the necessary items to the `Localizable.strings` file in appropriate places where you can find them again later:

```
/* Presented when telephone number is too short */
"Telephone number is too short" = "Telephone number is too short"

/* Informative text for alert posed by Telephone text field when incomplete
phone number is entered */
"The Telephone text field requires a complete telephone number of the form
“(800) 555-1212”." = "The Telephone text field requires a complete telephone
number of the form “(800) 555-1212”."
```

f. Now you can start writing the input filtering and formatting method. First, add the signature and the first statement in the method to `MyTelephoneFormatter.m`:

```
// Input filter and formatter

- (BOOL)isPartialStringValid:(NSString **)partialStringPtr
proposedSelectedRange:(NSRangePointer)proposedSelRangePtr
originalString:(NSString *)origString
originalSelectedRange:(NSRange)origSelRange errorDescription:(NSString
**)error;

    NSString *lastInsertedString = [*partialStringPtr
substringWithRange:NSMakeRange(origSelRange.location, proposedSelRangePtr-
>location - origSelRange.location)];
}
```

This is identical to the Decimal number formatter, obtaining the string that was input by the user either by typing one character or pasting one or more characters.

g. Next, filter the `lastInsertedString` variable for invalid characters to be rejected. Add the following immediately after the first statement you added in the previous instruction:

```
if ([lastInsertedString
rangeOfCharacterFromSet:invertedDecimalDigitPlusTelephoneSeparatorsCharacterSet
options:NSLiteralSearch].location != NSNotFound) {
    *error = [NSString stringWithFormat:NSLocalizedString(@"“%@” is not valid
in a telephone number", @"Presented when typed or pasted value contains a
character other than a numeric digit or telephone separator"),
lastInsertedString];
    return NO;
}
```

This code is also nearly identical to the Decimal number formatter.

h. The `errorDescription` parameter value should, of course, be localized. Add the following to the new Formatters section of the `Localizable.strings` file (this message isn't very smooth, but it will do for your purposes):

```
/* Presented when typed or pasted value contains a character other than a
numeric digit or telephone separator */
"“%@” is not valid in a telephone number" = "“%@” is not valid in a
telephone number"
```

i. Just as the Decimal number formatter required code to prevent the user from typing an unlimited number of decimal separator characters, the Telephone number formatter requires code to prevent the user from typing too many digits. A valid formatted telephone number should have ten digits and be fourteen characters long when formatted with separators.

Add the following after the block you just inserted in `MyTelephoneFormatter.m`:

```
if ([*partialStringPtr length] > 14) {
    *error = NSLocalizedString(@"Telephone number is too long", @"Presented
when telephone number is too long");
    return NO;
}
```

j. The `errorDescription` parameter value should, again, be localized. Add the following to the Formatters section of the `Localizable.strings` file:

```
/* Presented when telephone number is too long */
"Telephone number is too long" = "Telephone number is too long"
```

k. Adding automatic, on-the-fly telephone separators as the user types will be done using the same strategy you used in the Decimal number formatter: strip any existing telephone separators from the contents of the text field, then insert new telephone separators wherever needed, adjusting the insertion point backwards and forwards at every stage to keep it synchronized with the contents of the field.

As you will see, the fact that there are several different telephone separators requires you to use a character set rather than a string to hold the separators. In addition, since two of the separators always come as a pair, adjusting the insertion point requires some extra gymnastics.

First, add the following code at the end of the formatter's main method:

```
// Adjust location of telephone separators:

scanner = [NSScanner localizedScannerWithString:*partialStringPtr];
[scanner setCharactersToBeSkipped:[NSCharacterSet
characterSetWithCharactersInString:@""]];
*partialStringPtr = @"";
while (![scanner isAtEnd]) {
    NSString *tempString;
    if ([scanner scanUpToCharactersFromSet:telephoneSeparatorsCharacterSet
intoString:&tempString]) {
        *partialStringPtr = [*partialStringPtr
stringByAppendingString:tempString];
    } else if ([scanner
scanCharactersFromSet:telephoneSeparatorsCharacterSet
intoString:&tempString]) {
        if ([scanner scanLocation] <= (origSelRange.location +
[lastInsertedString length])) {
```

```
            proposedSelRangePtr->location = proposedSelRangePtr->location -
[tempString length];
        }
    }
}
```

You also have to declare the `scanner` variable, at the top of the formatter's main method, as follows:

```
NSScanner *scanner;
```

This block differs from the corresponding block in the Decimal number formatter, in that it does not scan telephone separators into oblivion immediately, but instead scans them into a temporary local string. It will discard the contents of the string as soon as it is done with them, but it uses them briefly first to determine how many separator characters were scanned at once in order to decrement the insertion point by that number of positions. As a result, when you use this formatter, you will observe that deleting or forward deleting over the ") " separator following the area code will cause the insertion point to jump two characters forward or backward.

l. Next, you must add telephone separators back into the text field in appropriate locations.

Add the following after the block you just completed:

```
for (index = 0; index <= [*partialStringPtr length]; index++) {
    NSString *delimiterString;
    switch (index) {
        case 0: delimiterString = @"("; break;
        case 4: delimiterString = @") "; break; // Space after close paren
        case 9: delimiterString = @"-"; break;
        default: delimiterString = @""; break; // Empty string
    }
    if ([delimiterString length] > 0) {
        *partialStringPtr = [NSString stringWithFormat:@"%@%@%@",
[*partialStringPtr substringToIndex:index], delimiterString,
[*partialStringPtr substringFromIndex:index]];
        if ((index <= proposedSelRangePtr->location) && !((lastTypedChar ==
NSDeleteCharacter) && ([telephoneSeparatorsCharacterSet
characterIsMember:[*partialStringPtr characterAtIndex:(proposedSelRangePtr-
>location + [delimiterString length] - 1)]]))) {
            proposedSelRangePtr->location = proposedSelRangePtr->location +
[delimiterString length];
        }
    }
}
```

This block ues a `for` loop to traverse the current contents of the text field (now stripped of its telephone separator characters), including the most recent inserted character or characters, one character at a time. It uses a switch statement to set the local variable `delimiterString` to whatever telephone separator string should be inserted at specific iterations. Note that the delimiter string is a two-character sequence at position 5. In iterations where no telephone separator belongs, the delimiter string is set to an empty string. The length of the delimiter string is used to determine when the insertion point requires adjustment and by how many character positions.

m. The special handling of the Delete key is almost the same as in the Decimal number formatter. Note that the incrementing of the insertion point is only preempted once because the user pressed the Delete key for any eligible separator, even if multiple separators have been inserted at once.

The code in instruction 5.l., above, used two variables that have not yet been declared, and this snippet of code: `(lastTypedChar == NSDeleteCharacter)` to detect the user's pressing of the Delete key. You now need to declare these variables and provide the code to make the delete character snippet work. At the top of the formatter's main method, after the

declaration of `scanner`, add the following:

```
int index;

unichar lastTypedChar = [[[[myControl window] currentEvent] characters]
characterAtIndex:0];
```

The character last typed by the user is obtained from the most recent system event that was generated by the user's typing activity in the window in which the control to which it is attached is located. This is different from how you obtained the most recent character typed in the Decimal number formatter. There, you used the application's `currentEvent` method; here, you use the control's window's `currentEvent` method. According to the documentation, the window method simply calls the application method, so there is no difference in effect. You used the window's method here only to see one way in which you could make use of the reference to the control to which this formatter is attached, which was passed into this formatter and set up in this formatter's designated initializer. In another formatter, you might actually need a reference to the control, or a reference to the window in which it resides, in order to make decisions about formatting issues that might depend upon the current state of the control or of the window. It would also give you access to the window's field editor, allowing the use of more advanced techniques for formatting the text field on the fly.

n. Finally, the method must return some values. Insert the following code at the end of the method:

```
if ([*partialStringPtr length] == 0) {
    [[[myControl window] fieldEditor:NO forObject:nil]
setSelectedRange:*proposedSelRangePtr];
}
error = nil;
return NO;
```

You will recognize the block of code that works around the bug you learned about in *Step 1*. This block is slightly different from the one you used there, however, making use of the `myControl` local variable to get at the text field's window, instead of using `[NSApp keyWindow]`.

You took care of connecting the Telephone text field in *Step 1*, so you can build and run the application now. Select the Text Fields pane and press the Tab key three times to move the insertion point to the Telephone text field, or click directly in it.

Type a telephone number and watch as telephone separators appear in the proper locations automatically. Use the Delete and Forward Delete keys and watch as the separators continue to be positioned correctly. Be sure to try selecting various portions of the number and deleting, forward deleting, typing, cutting and pasting. Typing or pasting a string containing illegal characters generates an error message. Note also that traditional means of selecting text work as they are supposed to, such as holding down the Shift key while using the arrow keys. Verify that all works as you would expect, and that the insertion point always ends up at a point that allows you to continue typing or to resume typing where a deletion occurred. Try typing or pasting a telephone separator explicitly, and you will find that, although no error is generated, the characters appear where they are supposed to, not necessarily where you tried to put them.

It is particularly interesting to place the insertion point immediately before or after the paired separator string ") ", then alternately hit the Delete and Dorward Delete keys. The insertion point moves back and forth to either side of the pair, without deleting it or either character in it. If you deliberately place the insertion point in the middle of the pair, between ")" and " ", then press the Delete or Forward Delete key, the insertion point will jump only a single character in order to get to the next numerical digit on either side.

Finally, hit the Enter key (or click on another text field or Tab out of the Telephone field) while a number with too few characters is in the field. Notice that an appropriate alert sheet is presented. On the other hand, if you delete the entire contents of the field and press Enter, you will be allowed to do that.

There is another on-the-fly telephone formatter available on the Web for comparison with the one you have just completed. The Omni Group, long-time masters of OpenStep and Cocoa programming, make their Omni Frameworks available for study and use by all. One of their frameworks, OmniFoundation, contains a large number of custom formatters, including OFTelephoneFormatter. It is worth taking a look at the Omni version, because it was written by overriding the old, pre-Mac OS X 10.0 `isPartialStringValid:newEditingString:errorDescription:` method, rather than the new, more powerful `isPartialStringValid:proposedSelectedRange:originalString:originalSelectedRange:errorDescription:`

method you used here. The difference is striking; your code is much shorter and simpler than that of the Omni version, thanks to the fact that Cocoa now does even more of the work for you.

You are now finished with the subject of custom formatters. In *Recipe 6*, one thing you will learn is that a single instance of a custom formatter can be used repeatedly by many different text fields within a window, thus conserving memory and making your code shorter. You will also see how your formatters work when text is dragged and dropped into a text field and when undo capability is turned on in a text field.