Erik M. Buck
Donald A. Yacktman

Foreword by Aaron Hillegass,
author of *Cocoa Programming for Mac OS X*

Cocoa
Design Patterns

Developer's Library

Library of Congress Cataloging-in-Publication Data:

Buck, Erik M.

Cocoa design patterns / Erik M. Buck, Donald A. Yacktman.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-53502-3 (pbk. : alk. paper) 1. Cocoa (Application development environment) 2. Object-oriented programming (Computer science) 3. Software patterns. 4. Mac OS. I. Yacktman, Donald A. II. Title.

QA76.64.B82 2009

005.26'8—dc22

2009023288

ISBN-13: 978-0-321-53502-3
ISBN-10: 0-321-53502-2
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing September 2009

Editor-in-Chief
Karen Gettman

Acquisitions Editor
Chuck Toporek

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Language Logistics, LLC

Indexer
Rebecca Salerno

Proofreader
Apostrophe Editing Services

Publishing Coordinator
Romny French

Cover Designer
Gary Adair

Compositor
Jake McFarland

Foreword

Grumpy old men are represented poorly by our modern culture. They are always depicted throwing stuff and bellowing lines like, "Hey, you kids, get off my lawn!" In reality, grumpy old men often say useful things like, "Kid, you should diversify your portfolio—just in case."

As someone who has been developing applications with Cocoa and Objective-C for a long time, one of my important roles is that of a grumpy old man. Programmers who are new to Cocoa come to me and say things like, "Here's my program. It works great. You want to look at the source?"

And I study the source code and growl things like, "Yes, that works, but that isn't how we do it. We grumpy old Cocoa programmers have a system worked out, and you are not following the system."

And the young programmer says, "Well, why is your system so great?"

And I grunt, "Um...well...it just is! Shut up and get off my lawn, kid."

The book you are holding is the answer to two important questions:

- How do the grumpy old Cocoa programmers do things?
- Why is that so great?

Through floundering about with bad solutions, grumpy old Cocoa programmers have figured out some really good solutions to common design problems. The existence of this book means that you are not required to suffer through the same misery that we went through.

Both Erik M. Buck and Donald A. Yacktman have earned their grumpy, old Cocoa programmer status. They each have enough successes and enough failures to recognize what a good Cocoa design looks like. Beyond presenting these idioms and techniques, Erik and Donald have included serious meditations on why it was these patterns emerged from the chaos that was Objective-C programming a decade ago.

Next time some kid shows up at my door asking for a code review, *this* is the book I am going to throw at him. It is a pity there is no hardcover edition.

—*Aaron Hillegass*
Big Nerd Ranch, Inc.
Atlanta, Georgia

Preface

Much of the technology embodied by Apple's Cocoa software has been in commercial use since 1988, and in spite of that maturity, Cocoa is still revolutionary. The technology has been marketed with a variety of names including NEXTSTEP, OPENSTEP, Rhapsody, and Yellow Box. It consists of a collection of reusable software frameworks that contain objects and related resources for building Mac OS X desktop and mobile applications. In recent years, Apple has expanded Cocoa dramatically and added new software developer tools to increase programmer productivity beyond the already famously high levels Cocoa already provided.

Programmers are often overwhelmed by the breadth and sophistication of Cocoa when they first start using the frameworks. Cocoa encompasses a huge set of features, but it's also elegant in its consistency. That consistency results from the application of patterns throughout Cocoa's design. Understanding the patterns enables the most effective use of the frameworks and serves as a guide for writing your own applications.

This book explains the object-oriented design patterns found in Apple's Cocoa frameworks. Design patterns aren't unique to Cocoa; they're recognized in many reusable software libraries and available in any software development environment. Design patterns identify recurring software problems and best practices for solving them. The primary goal of this book is to supply insight into the design and rationale of Cocoa, but with that insight, you'll be able to effectively reuse the tried and true patterns in your own software—even if you aren't using Cocoa.

What Is a Design Pattern?

Design patterns describe high quality practical solutions to recurring programming problems. Design patterns don't require amazing programming tricks. They're a toolbox of reusable solutions and best practices that have been refined over many years into a succinct format. They provide a vocabulary, or shorthand, that programmers can use when explaining complex software to each other. Design patterns don't describe specific algorithms or data structures like linked lists or variable length arrays, which are traditionally implemented in individual classes. The design patterns in this book don't describe specific designs for applications even though examples are provided. What the patterns do provide is a coherent map that leads you through the design of Cocoa itself. Patterns show how and why some of the best and most reusable software ever created was designed the way it was.

At a minimum, design patterns contain four essential elements:

- The pattern name
- A brief description of the motivation for the pattern or the problem solved by the pattern
- A detailed description of the pattern and examples in Cocoa
- The consequences of using the pattern

Parts II, III, and IV of this book contain a catalog of design patterns. Each chapter in the pattern catalog introduces a design pattern and provides the essential information you need to recognize and reuse the pattern.

The pattern's name helps developers communicate efficiently. A shared vocabulary of pattern names is invaluable when explaining a system to colleagues or writing design documentation. Named patterns clarify thought, and the implications of a design—even the rationale behind a design—can be communicated with just a few words.

Programmers familiar with patterns immediately infer the uses and limitations of objects composing a named pattern as well as the overall design employed and the consequences of that design.

Apple's own documentation occasionally uses design pattern names in both class references and programmer's guides, but the documentation doesn't always explain what the patterns are or what they should mean to a developer. In addition, Apple frequently uses its own names for design patterns instead of the terms commonly used throughout the industry. In some cases, the differences in terminology are the result of simultaneous independent discovery. In other cases, the patterns were first recognized in Cocoa or its predecessor NEXTSTEP, and it's the industry that changed the name. The patterns described in this book are identified using both Apple's terminology and the common industry names when applicable so you can see the correlation.

Each design pattern includes a description of the problem(s) and motivation for applying the pattern. Some patterns include a list of problem indicators that suggest the use of the pattern. Because Cocoa contains many patterns that are applicable in diverse situations, the patterns have been carefully organized so that the same problems in different contexts are readily identified. In some cases, related patterns that should be avoided are also identified.

Finally, each pattern identifies the consequences that naturally result from its use. The consequences and trade-offs of design alternatives are crucial when evaluating which patterns to use in a particular situation.

Why Focus on Design Patterns?

When approaching a software technology as vast as Cocoa, it's easy to lose sight of the overall architecture and rationale of the technology. Many programmers comment that they feel lost in the multitude of classes, functions, and data structures that Cocoa provides. They can't see the forest because they're concentrating too much on individual

trees. The patterns used in Cocoa provide a structure and organization that helps programmers find their way. The patterns show programmers how to reuse groups of cooperating classes even when the relationships between the classes are not fully explained in the documentation for individual classes.

The goal of object-oriented programming is to maximize programmer productivity by reducing lifetime software development and maintenance costs. The principal technique used to achieve the goal is object reuse. An object that is reused saves the programmer time because the object would otherwise need to be reimplemented for each new project. Another benefit of reusing objects is that when new features are required or bugs are identified, you only need to make changes to a small number of objects, and those changes benefit other projects that rely on the same objects. Most importantly, by reusing objects, fewer total lines of code are written to solve each new problem, and that means there are fewer lines of code to maintain as well.

Design patterns identify successful strategies for achieving reuse on a larger scale than individual objects. The patterns themselves and all of the objects involved in the patterns are proven and have been reused many times. The consistent use of design patterns within Cocoa contributes to the high level of productivity that Cocoa programmers enjoy. Design patterns advance the art of object-oriented programming.

The patterns within Cocoa provide a guide for designing many different types of applications. Cocoa contains some of the most famously well-designed software ever produced, and following the patterns used by Cocoa will make you a better programmer even when you aren't using Cocoa.

This book should satisfy your intellectual curiosity. Design patterns answer "why" as well as "what" and "how." Knowing how patterns are applied and more importantly why patterns contribute so much to productivity makes the daily job of programming more enjoyable.

Guiding Principles of Design

All of the design patterns described in this book have several properties in common. In each case, the goal of the pattern is to solve a problem in a general, reusable way. Several guiding principles of design help ensure that the patterns are flexible and applicable in many contexts. The same strategies that are applied to the design of individual objects are applied to design patterns as well. In fact, patterns that involve many objects benefit even more from good object-oriented design than simpler systems. One reason that patterns exist is to help make sure that productivity gained from reusing the patterns exceeds the productivity gained from using individual objects—the sum is greater than the parts.

Minimize Coupling

As a general design goal, coupling between classes should be minimized. Coupling refers to dependencies between objects. Whenever such dependencies exist, they reduce opportunities for reusing the objects independently. Coupling also applies to subsystems within

large systems of objects. It's important to look for designs that avoid coupling whenever possible.

All of the Cocoa design patterns exist in part to limit or avoid coupling. For example, the overarching Model-View-Controller (MVC) pattern described in Part I of this book, "One Pattern to Rule Them All," is used throughout Cocoa to organize subsystems of classes and is applied to the design of entire applications. The primary intent of the MVC pattern is to partition a complex system of objects into three major subsystems and minimize coupling between the subsystems.

Design for Change

It's important to use designs that accommodate changes through the lifecycle of a software system. Designs that are too inflexible ultimately restrict opportunities for reuse. In the worst case, no reuse occurs because it's easier to redesign and reimplement a system than it is to make changes within an existing rigid design.

It's possible to anticipate certain types of changes and accommodate them in a design. For example, the Cocoa Delegates pattern provides a mechanism for one object to modify and control the behavior of another object without introducing coupling between them. Cocoa provides many objects that can be controlled by optional delegates, and the key to the pattern is that the objects acting as delegates might not have even been conceived when Cocoa was designed. All of the Cocoa design patterns exist in part to accommodate change. That's just one of the reasons that Cocoa is so flexible.

Emphasize Interfaces Rather Than Implementations

Interfaces provide a kind of metaphorical contract between an object and the users of the object. An object's interface tells a programmer what the object is able to do but not how it will do it. In the context of reusable frameworks like Cocoa, object interfaces must remain consistent from one version of the framework to the next, or else software written to use one version of the framework may not work correctly with the next. A contract is necessary for programmers to feel confident reusing framework objects, but anyone who has tried to create a truly flexible reusable contract knows that it's a difficult task. When implementation details become part of the contract between an object and its users, it becomes difficult for framework developers to improve objects without breaking backward compatibility.

Find the Optimal Granularity

Many of the design patterns employed by Cocoa operate at different levels of granularity. For example, the MVC pattern is usually applied to large subsystems of cooperating classes and entire applications, but the Singleton pattern is used to make sure that only one instance of a class is ever created and provides access to that instance. The goal of patterns is to enhance software reuse. The granularity of a pattern can have a huge impact on opportunities for reuse.

Certain problems are best solved by small patterns that involve only a few classes, while other problems are solved by reusing grand overarching patterns. The key is to find the optimal balance. In general, the larger patterns provide bigger productivity gains than the smaller ones, but if a pattern is too big or too general to solve a specific, narrow problem, it can't be used. For example, the MVC pattern contributes enormously to most applications, but there are some specific applications that may not benefit from its use, and in those cases the pattern provides no value. In contrast, patterns such as Anonymous Objects and Heterogeneous Containers, Enumerators, Flyweight, and Singleton are small and contribute value in every application. Cocoa provides patterns all along the spectrum. Some of the pattern descriptions address the issues of granularity and the balance that Cocoa strikes.

Use Composition in Preference to Inheritance

It can't be said enough times that coupling is the enemy. It is ironic that inheritance is simultaneously one of the most powerful tools in object-oriented programming and one of the leading causes of coupling. In fact, there is no tighter coupling than the relationship between a class and its inherited superclasses. Many of the patterns described in this book exist in part to reduce the need to create subclasses. The general rule is that when there is an alternative to inheritance, use the alternative.

Audience

This book is intended for Mac OS X programmers who are using or considering the use of Apple's Cocoa frameworks for Mac OS X or the Cocoa Touch frameworks for iPhone and iPod Touch. Much of the information in this book also applies directly to the open source GNUstep project, which is available for Linux and Windows.

Who Should Read This Book

Objective-C, C, C++, and Java programmers should read this book. You should be familiar with the general principals of object-oriented design and object-oriented technology to understand and benefit from the design patterns presented here. Many of Cocoa's design patterns leverage features of the Objective-C language, which are not thoroughly explained in this book; however, Apple includes the document, titled *The Objective-C 2.0 Programming Language*, along with the free Mac OS Xcode Tools (http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC).

Some knowledge of Objective-C is required to understand the implementation of Cocoa, although experienced programmers can pick it up incrementally while reading this book. That said, this book is not a substitute for a language reference such as *The Objective-C 2.0 Programming Language* even though language features that contribute to Cocoa design patterns are explained as needed within the pattern descriptions.

What You Need to Know

This book doesn't require guru-level programming skills. The patterns used in the design of Cocoa are identified and explained in part to demystify the technology. Programmers who are new to Cocoa will benefit from the insights and wisdom embodied by Cocoa just as much as experienced veterans. However, if you are completely new to programming with C or languages derived from C, you'll have difficulty following the in-depth analysis of how and why patterns work. You need to be comfortable with the object-oriented concepts of classes, instances, encapsulation, polymorphism, and inheritance. Without a foundation in the technology of object-oriented software development, the sometimes advanced descriptions of benefits, consequences, and trade-offs in this book could be overwhelming.

This book assumes that you know C, C++, or Java and that you're familiar with object-oriented software development. As mentioned earlier, you need to know Objective-C to get the most value from this book, but Objective-C can be learned along the way.

You need to be running a Mac OS X system with Apple's Xcode Tools installed. If you don't have the Xcode Tools installed on your system, there are a couple things you can do to obtain them:

- If you purchased new Mac hardware or a boxed release of Mac OS X, the Xcode Tools can be found on the Install DVD in the Optional Installs folder.
    - For Mac OS X Leopard (v 10.5), look in the Xcode Tools folder and double-click on the XcodeTools.mpkg file to install Xcode.
    - For Mac OS X Snow Leopard (v 10.6), double-click on the Xcode.mpkg file to install Xcode.
- The latest Xcode Tools are available with a free online membership to the Mac Developer Program which is part of the Apple Developer Connection (ADC) at http://developer.apple.com/. After you've signed up, you can download the Xcode Tools from the ADC website. (Keep in mind, though, that the download is around 1GB, so you'll need a fast connection.)

> **Note**
>
> If you are developing for the iPhone or iPod Touch, you need to register for the iPhone Developer Program (http://developer.apple.com/iphone), and then download and install the iPhone Software Development Kit (SDK) from the iPhone Dev Center after logging in. The iPhone 3.0 SDK requires an Intel-based Mac which means it cannot be used on older, PowerPC-based Macs (for example, Macs with the G3, G4, or G5 processors). The iPhone SDK is available for Mac OS X Leopard (v 10.5) and for Mac OS X Snow Leopard (v 10.6).

This book is written based on Mac OS X (v 10.5), but ultimately you will want to leverage Cocoa's design patterns when creating applications for any version of Mac OS X, iPhone, iPod Touch, or for Windows and Linux with GNUstep.

How This Book Is Organized

This book is organized into five parts. Part I, "One Pattern to Rule Them All," describes the Model-View-Controller pattern that provides the overall structure and organization for Cocoa and most applications that use Cocoa. Part II, "Fundamental Patterns," identifies the patterns in Cocoa with which all other patterns are built. Part III, "Patterns That Primarily Empower by Decoupling," contains patterns that enable you to control and extend objects without introducing unnecessary coupling. Part IV, "Patterns That Primarily Hide Complexity," explains patterns that hide complexity and implementation details so programmers can confidently focus on solving problems. Part V, "Practical Tools for Pattern Application," shows practical applications of the Model-View-Controller design pattern with examples selected from the Cocoa frameworks. Appendix, "Resources," provides additional references for using and understanding Cocoa and design patterns.

29

Controllers

Within the overarching Model View Controller (MVC) design pattern, the Controller subsystem has historically lagged the other subsystems when it comes to object reuse. Controller subsystems are full of "glue" code used to mediate between views and models. In contrast, the Model and View subsystems are replete with standard reuse scenarios. Mature flexible data structures, databases, and algorithms for models were well established decades ago. Standard reusable view objects shipped with the earliest ancestors of Cocoa in 1988 complete with Interface Builder. The Application Kit leverages patterns to almost automate the development of View subsystems. But what about controllers? How are design patterns applied to simplify controllers, promote wide scale controller reuse, and automate controller development? The Controller subsystem in Cocoa has only lately matured and standardized, and the only real explanation for the delay is that it has taken longer to recognize the design patterns that are applicable for controllers.

Consider how controllers differ from views. Conventions and metaphors for user interaction with views are now standard. For example, users understand the concept of the "current selection" within a user interface and that using the "Copy" menu item will copy the current selection and not some other part of the interface. The metaphors and conventions for views had to be established before design patterns like Cocoa's Responder Chain were applied to implement those conventions. Conventions and metaphors for controllers are less clear. Controllers integrate views with models as diverse as games, employee benefits management, weather simulations, and robotic arm manipulation. This chapter explores some common controller tasks and identifies opportunities for reuse in "glue" code. In the process, this chapter exposes the rationale for the various Cocoa `NSController` subclasses and the resulting design patterns.

Motivation

Reduce the need for recurring error prone code when implementing the Controller subsystem of the Model View Controller design pattern. Apply Apple's Interface Builder tool and the Controllers pattern to streamline development of the Controller subsystem for simple applications and substantially reduce the code needed to implement complex applications.

Solution

This section presents the relatively simple MVC `MYShapeDraw` application example shown in Figure 29.1. The example highlights typical tasks a Controller subsystem needs to perform. Initially, the entire implementation of `MYShapeDraw`'s controller subsystem is in just one class. The example includes the kind of code that has historically been written and rewritten for almost every MVC application. Once the `MYShapeDraw` application is fully developed, the example's controller is redesigned to make it more general and reusable. By the end of this section, the example's Controller subsystem evolves into a clone of the design used by Cocoa's `NSArrayController` class. Following the step-by-step reinvention of `NSArrayController` in this chapter reveals why Cocoa's `NSObjectController` and its subclasses exist and how they're used in applications.

Figure 29.1 The user interface for `MYShapeDraw` application

The MYShapeDraw example application has the following features/requirements above and beyond the features provided by all Cocoa document-based applications:

- Provide a simple Model subsystem: just an array of shape objects.
- Provide a custom graphical view to display shape objects.
- Provide a way to add shape objects to the model.
- Provide a way to select zero, one, or multiple shape objects.
- Provide a way to reposition selected shape objects in the custom view.
- Provide a way to remove selected shape objects from the model.
- Provide a table view to display information about shape objects.
- When either the model or any of the views change, update the others.

There's a lot of code in this section because controllers can't be analyzed in isolation. It's necessary to develop a minimal model and view just to see how the controller interacts. Some of the code for the Model and View subsystems is omitted from this chapter for the sake of brevity and to keep the focus on the Controller subsystem. All of the code is available at www.CocoaDesignPatterns.com.

MYShapeDraw Model Subsystem

The model for this example is just an array of MYShape instances. The MYShape class encapsulates a color and a rectangle that defines the shape's position and size.

> **Note**
>
> The model in this example is deliberately kept simple to preserve the focus on the Controller subsystem. In most applications, properties like rectangles and colors are user interface concerns that don't belong in the Model subsystem. However, in this case, MYShapeDraw is a drawing program. The objects that the user wants to view or edit are shapes. Imagine that the shapes edited by MYShapeDraw actually represent the holes to be cut out of a sheet of metal and the colors represent the color of the wires to be routed through the holes. The model then consists of instructions to be sent to cutting and wire routing machines.

A more full-featured Model subsystem might include subclasses of MYShape to represent circles, text, images, and groups of shapes. However, the following base MYShape class is sufficient for this example:

```
@interface MYShape : NSObject <NSCoding>
{
  NSRect        frame;
  NSColor       *color;
}

@property (readwrite, assign) CGFloat positionX;
@property (readwrite, assign) CGFloat positionY;
```

```
@property (readwrite, copy) NSColor *color;

// Returns the receiver's frame
- (NSRect)frame;

// Moves the receiver's frame by the specified amounts
- (void)moveByDeltaX:(float)deltaX deltaY:(float)deltaY;

// This is a Template Method to customize selection logic.  The default
// implementation returns YES if aPoint is within frame.  Override this
// method to be more selective.  The default implementation can be
// called from overridden versions.
- (BOOL)doesContainPoint:(NSPoint)aPoint;

@end
```

The properties declared for the `MYShape` class are not identical to the instance variables declared for the class. There's no particular reason for properties and instance variables to coincide, and it's convenient for this example to provide `positionX` and `positionY` properties. The Accessor methods (see Chapter 10, "Accessors") for the properties are implemented to calculate values relative to the frame. The implementation of the `MYShape` class is so simple that it doesn't need to be shown here, but it's available in the example source code.

`MYShapeDraw` View Subsystem

Based on the requirements for this example, there are at least two different ways to view and interact with the model. A custom `NSView` subclass is needed to display and select shapes and enable graphical repositioning of selected shapes. An ordinary `NSTableView` is needed to display information about shapes in a table.

This example doesn't require any code in the View subsystem to use a `NSTableView`. All of the table configuration is performed in Interface Builder, and the upcoming Controller subsystem provides the data the table needs.

Implementing the custom `NSView` subclass is almost as straightforward as the model. To start, declare the `MYShapeView` subclass of `NSView` as follows:

```
@interface MYShapeView : NSView
{
  IBOutlet id        dataSource;
}

@property (readwrite, assign) id dataSource;  // Don't retain or copy

@end
```

No new methods are needed. The entire functionality of `MYShapeView` is either inherited from the `NSView` class, overridden from the `NSView` class, or provided by the one and only property, `dataSource`. The `dataSource` is used to implement the Data Source

pattern explained in Chapter 15, "Delegates." `MYShapeView` instances interrogate their data sources to determine what to draw. The `MYShapeView` is implemented as follows:

```
@implementation MYShapeView

@synthesize dataSource;


- (void)dealloc
{
  [self setDataSource:nil];
  [super dealloc];
}


// Draw all of the MYShape instances provided by the dataSource
// from back to front
- (void)drawRect:(NSRect)aRect
{
  [[NSColor whiteColor] set];
  NSRectFill(aRect);  // Erase the background

  for(MYShape *currentShape in
      [[self dataSource] shapesInOrderBackToFront])
  {
    [currentShape drawRect:aRect];
  }
}

@end
```

That's pretty much all it takes to draw shapes. `MYShapeView` overrides `NSView`'s `-drawRect:` Template Method to get an array of `MYShape` instances from the `dataSource` and then send a message to each shape requesting that it draw itself. Template Methods are explained in Chapter 4, "Template Method." An interesting question arises at this point: How do `MYShape` instances know how to draw themselves in `MYShapeView` instances? Drawing is clearly part of the View subsystem, but the `MYShape` class is declared in the Model subsystem. The solution used in this example applies the Category pattern from Chapter 6, "Category," to extend the `MYShape` class within the View subsystem using the following declaration and implementation:

```
// Declare an informal protocol that MYShape instances must implement
// in order to be displayed in a MYShapeView.
@interface MYShape (MYShapeQuartzDrawing)

// This is a Template Method to customize drawing.  The default
// implementation fills the receiver's frame with the receiver's color.
// Override this method to customize drawing.  The default
```

```
// implementation can be called from overridden versions, but it is
// not necessary to call the default version.
- (void)drawRect:(NSRect)aRect;

@end


@implementation MYShape (MYShapeQuartzDrawing)

// Draw the receiver in the current Quartz graphics context
- (void)drawRect:(NSRect)aRect
{
  if(NSIntersectsRect(aRect, [self frame]))
  {
    [[self color] set];
    NSRectFill([self frame]);
  }
}

@end
```

The `MYShapeQuartzDrawing` category is implemented right in the same file as the `MYShapeView` class. Therefore, all of the relevant code for drawing `MYShape` instances in `MYShapeViews` is maintained together.

> **Note**
>
> A future `MYShapeOpenGLView` might draw `MYShape` instances using Open GL instead of Quartz. The `MYShapeOpenGLView` class could provide its own category of `MYShape` to add a `-drawRect:(NSRect)aRect forOpenGLContext:(NSOpenGLContext *)aContext` method. In that way, the Open GL-specific drawing code could be maintained right next to the rest of the `MYShapeOpenGLView` code.

The `MYShapeView` class provides basic display of the `MYShape` instances supplied by a `dataSource`. The code to support graphical editing features could be added to the `MYShapeView` class, but sometimes it's handy to have a simple display-only class like `MYShapeView`. The graphical editing support will be added in a subclass of `MYShapeView` called `MYEditorShapeView` later in the example, but for now, `MYShapeView` provides enough capability to move on to the Controller subsystem.

`MYShapeEditor` Controller Subsystem

So now that the model and view are established, what does the Controller subsystem need to do? The Controller subsystem needs to initialize the model either from scratch or by loading a previously saved model. The Controller subsystem must set up the view. The Controller subsystem must supply an object that will serve as the table view's data source and an object that will serve as the custom view's data source. The Controller subsystem must enable adding shapes to the model. The Controller subsystem needs to keep track of

which shapes are selected and enable removal of selected shapes from the model. Finally, the Controller subsystem must keep the model and all views up to date.

The list of controller tasks fall into two general categories, coordinating tasks and mediating tasks. Coordinating tasks include loading the Model and View subsystems and providing data sources. Mediating tasks control the flow of data between view objects and model objects to minimize coupling between the subsystems, while keeping them synchronized.

Coordinating Controller Tasks

The first step in the implementation of `MYShapeEditor`'s Controller subsystem is to tackle the coordinating tasks. Almost every MVC application must set up a view and initialize a model, and the Cocoa framework provides the `NSDocument` class for just that purpose. `NSDocument` declares the `-windowNibName` Template Method, which allows subclasses to identify an Interface Builder file containing the view objects to be loaded. The `-dataOfType:error:` and `-readFromData:ofType:error:` Template Methods support saving and loading model data. There are alternative, more sophisticated ways to use `NSDocument`, but those three methods are a good fit for this example.

Create a `MYShapeEditorDocument` subclass of `NSDocument`, provide a pointer to the array of shapes that will comprise the model, and override the necessary `NSDocument` methods. The following is just the starting point; it will be fleshed out as the example progresses:

```
@interface MYShapeEditorDocument : NSDocument
{
  NSArray              *shapesInOrderBackToFront; // The model
}

@property (readonly, copy) NSArray *shapesInOrderBackToFront;

@end
```

In the implementation of the `MYShapeEditorDocument` class, the `shapesInOrderBackToFront` property is redeclared as `readwrite` in a class extension also known as an *unnamed category* so that when the property is synthesized, a "set" Accessor method will be generated.

```
@interface MYShapeEditorDocument ()

@property (readwrite, copy) NSArray *shapesInOrderBackToFront;

@end
```

The following implementation of `MYShapeEditorDocument` takes care of the basic model and view creation:

```
@implementation MYShapeEditorDocument

@synthesize shapesInOrderBackToFront;
```

```
- (NSString *)windowNibName
{ // Identify the nib that contains archived View subsystem objects
  return @"MYShapeEditorDocument";
}


- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError
{ // Provide data containing archived model objects for document save
  NSData    *result = [NSKeyedArchiver archivedDataWithRootObject:
      [self shapesInOrderBackToFront]];

  if ((nil == result) && (NULL != outError))
  { // Report failure to archive the model data
    *outError = [NSError errorWithDomain:NSOSStatusErrorDomain
        code:unimpErr userInfo:NULL];
  }

  return result;
}


- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName
    error:(NSError **)outError
{ // Unarchive the model objects from the loaded data
  NSArray    *loadedShapes = [NSKeyedUnarchiver
      unarchiveObjectWithData:data];

  if(nil != loadedShapes)
  {
    [self setShapesInOrderBackToFront:loadedShapes];
  }
  else if ( NULL != outError)
  { // Report failure to unarchive the model from provided data
    *outError = [NSError errorWithDomain:NSOSStatusErrorDomain
        code:unimpErr userInfo:NULL];
  }

  return YES;
}

@end
```

The `-dataOfType:error:` method is called by `NSDocument` as an intermediate step in the sequence of operations to save the document to a file. `MYShapeEditorDocument` archives the model, an array of shapes, using the Archiving and Unarchiving pattern from

Chapter 11 and then returns the resulting NSData instance to be saved. The -readFromData:ofType:error: method is called by NSDocument when a previously saved document is loaded. MYShapeEditorDocument unarchives an array of shapes from the provided data. The -windowNibName method returns the name of the Interface Builder .nib file that contains an archive of the objects that compose the View subsystem. NSDocument unarchives the user interface objects in the named .nib file so they can be displayed on screen.

That's all it takes to specialize the inherited NSDocument behavior for loading the example's document interface and saving/loading the model. However, it's still necessary to create an array to store shapes when a new empty document is created. It's also necessary to clean up memory when documents are deallocated.

NSDocument's -windowControllerDidLoadNib: Template Method is automatically called after all objects have been unarchived from the document's .nib file but before any of the objects from the .nib are displayed. If no array of shapes has been created by the time -windowControllerDidLoadNib: is called, the following implementation of -windowControllerDidLoadNib: creates an empty array of shapes to use as the model:

```
- (void)windowControllerDidLoadNib:(NSWindowController *)aController
{
  [super windowControllerDidLoadNib:aController];

  if(nil == [self shapesInOrderBackToFront])
  { // Create an empty model if there is no other available
    [self setShapesInOrderBackToFront:[NSArray array]];
  }
}
```

MYShapeEditorDocument's -dealloc method sets the array of shapes to nil thus releasing the model when the document is deallocated.

```
- (void)dealloc
{
  [self setShapesInOrderBackToFront:nil];
  [super dealloc];
}
```

NSDocument is one of the most prominent controller classes in Cocoa. NSDocument provides lots of features that aren't directly relevant to this example including management of the document window's title, access to undo and redo support, periodic auto-save operations, printing, and other standard Cocoa features. NSDocument is straightforward to use, and there are similar document classes in other object-oriented user interface frameworks. NSDocument encapsulates most of the coordinating controller features of any multidocument application and leverages Template Methods extensively to enable customization.

Mediating Controller Tasks (Providing Information to Views)

Cocoa provides several mediating controller classes, and once you understand the roles they can play in your design, they're as easy to reuse as the `NSDocument` class. However, the reuse opportunities for mediator code aren't always readily apparent. For one thing, every application has a unique model and a different view, so how can the code that glues the different subsystems together be reused in other applications? To answer that question, the example implements specific mediator code to meet the application's requirements and then explores how that code is made reusable.

> **Note**
>
> The examples in this chapter progressively re-create Cocoa's `NSArrayController` class. The examples refactor `MYShapeEditor`'s design to reveal why `NSArrayController` and other Cocoa mediating controllers exist. Follow the sequence of changes to `MYShapeEditor` to see how reusable mediating controllers work.

To get started and keep the design simple, implement all of the custom mediation code for `MYShapeEditor`'s Controller subsystem right in the `MYShapeEditorDocument` class. Figure 29.2 illustrates the design.

Each `MYShapeEditorDocument` instance acts as the data source for an associated custom graphic view and the associated table view. `MYEditorShapeView` only has one data source method, `-shapesInOrderBackToFront`, and that's already provided by the `@synthesize` directive for `MYShapeEditorDocument`'s `shapesInOrderBackToFront` property. The `NSTableView` class requires its data source to implement `-numberOfRowsInTableView:` and `-tableView:objectValueForTableColumn:row:`, so those methods are added to the implementation of `MYShapeEditorDocument` as follows:

```
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
  return [[self shapesInOrderBackToFront] count];
}


- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
    row:(int)rowIndex
{
  id shape = [[self shapesInOrderBackToFront] objectAtIndex:rowIndex];

  return [shape valueForKey:[aTableColumn identifier]];
}
```

To enable editing in the table view, `MYShapeEditorDocument` needs to implement the `-tableView:setObjectValue:forTableColumn:row:` method.

```
- (void)tableView:(NSTableView *)aTableView setObjectValue:(id)anObject
   forTableColumn:(NSTableColumn *)aTableColumn row:(NSInteger)rowIndex
```

```
{
  [self controllerDidBeginEditing];
  id shape = [[self shapesInOrderBackToFront] objectAtIndex:rowIndex];

  [shape setValue:anObject forKey:[aTableColumn identifier]];

  [self controllerDidEndEditing];
}
```

Figure 29.2 The initial design for the MYShapeDraw application.

The `-controllerDidBeginEditing` and `-controllerDidEndEditing` methods (shown in bold within the implementation of `-tableView:setObjectValue:forTableColumn:row:`) are called before and after a shape is modified. Shapes are part of the model. `MYShapeEditorDocument` consolidates the code for synchronizing the model, the table view, and the custom view into just the `-controllerDidBeginEditing` and `-controllerDidEndEditing` methods so that as long as those methods are called before and after a change to the model, everything is kept updated.

The -controllerDidBeginEditing and -controllerDidEndEditing methods are declared in the following informal protocol, a category of the NSObject base class:

```
@interface NSObject (MYShapeEditorDocumentEditing)

- (void)controllerDidBeginEditing;
- (void)controllerDidEndEditing;

@end
```

The informal protocol means that MYShapeEditorDocumentEditing messages can safely be sent to any object descended from NSObject. Informal protocols are explained in Chapter 6.

MYShapeEditorDocument overrides its inherited -controllerDidEndEditing implementation with the following code:

```
- (void)controllerDidEndEditing
{
  [[self shapeGraphicView] setNeedsDisplay:YES];
  [[self shapeTableView] reloadData];
}
```

MYShapeEditorDocument's -controllerEndEditing method tells shapeGraphicView to redisplay itself at the next opportunity and tells shapeTableView to reload itself from its data source, which indirectly causes shapeTableView to redisplay itself, too. In order for -controllerEndEditing to work, Interface Builder outlets for shapeGraphicView and shapeTableView are needed. Therefore, the MYShapeEditorDocument class interface is updated to the following, and the connections to the outlets are made in Interface Builder to match Figure 29.3.

```
@interface MYShapeEditorDocument : NSDocument
{
  NSArray              *shapesInOrderBackToFront; // The model
  IBOutlet NSView      *shapeGraphicView;
  IBOutlet NSTableView *shapeTableView;
}

@property (readonly, copy) NSArray *shapesInOrderBackToFront;
@property (readwrite, retain) NSView *shapeGraphicView;
@property (readwrite, retain) NSTableView *shapeTableView;

@end
```

Add the corresponding @synthesize directives to the MYShapeEditorDocument implementation:

```
@synthesize shapeGraphicView;
@synthesize shapeTableView;
```

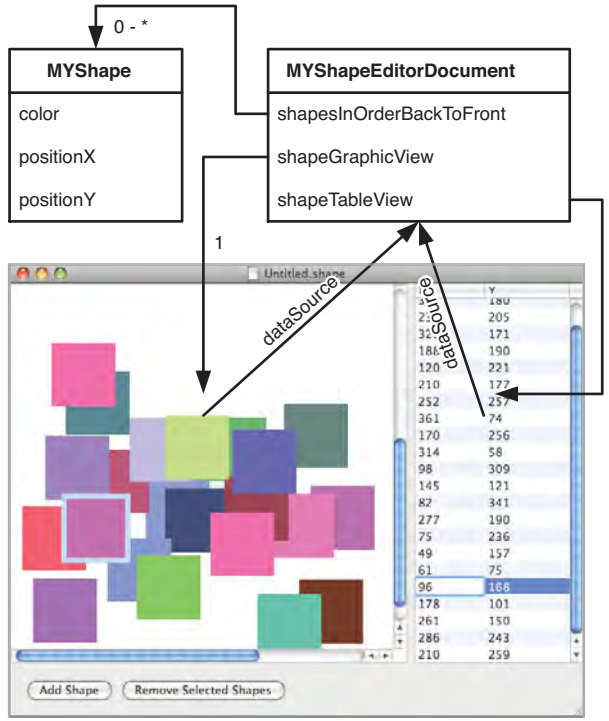

Figure 29.3 `MYShapeEditorDocument` outlets enable update of the views.

At this point, the example has produced a bare-bones shape viewer application with minimal shape editing support provided by the table view. The `MYShapeEditor0` folder at www.CocoaDesignPatterns.com contains an Xcode project with all of the code so far. Build the project and run the resulting application. Use the application to open the Sample.shape document provided at the same site. You can double-click the X and Y coordinates displayed in the table view to reposition the shapes in the custom view.

Mediating Controller Tasks (Selection Management)

The next feature to add to the Controller subsystem is the ability to keep track of the selected shapes in each document. One question to ask is whether keeping track of the selection is really a controller task at all, or should views perform that function? Storing selection information in the controller enables designs like the one for `MYShapeEditor` in which multiple views present information about the same model, and selection changes made in one view are reflected in the other views. The Consequences section of this chapter explains how storing selection information in the controller still makes sense even when multiple views have independent selections. Add an instance variable to store the

indexes of the selected shapes and selection methods to produce the following `MYShapeEditorDocument` interface:

```
@interface MYShapeEditorDocument : NSDocument
{
  NSArray              *shapesInOrderBackToFront;// The model
  IBOutlet NSView      *shapeGraphicView;
  IBOutlet NSTableView *shapeTableView;
  NSIndexSet           *selectionIndexes;        // selection
}

@property (readonly, copy) NSArray *shapesInOrderBackToFront;
@property (readwrite, nonatomic, retain) NSView *shapeGraphicView;
@property (readwrite, nonatomic, retain) NSTableView *shapeTableView;

// Selection Management
- (BOOL)setShapeSelectionIndexes:(NSIndexSet *)indexes;
- (NSIndexSet *)shapeSelectionIndexes;
- (BOOL)addShapeSelectionIndexes:(NSIndexSet *)indexes;
- (BOOL)removeShapeSelectionIndexes:(NSIndexSet *)indexes;
- (NSArray *)selectedShapes;

@end
```

The `selectionIndexes` variable uses an immutable `NSIndexSet` to efficiently identify which shapes are selected. Each `MYShape` instance in a document can be uniquely identified by the shape's index (position) within the ordered `shapesInOrderBackToFront` array. If a shape is selected, add the index of the selected shape to `selectionIndexes`. To deselect a shape, remove its index from `selectionIndexes`. To determine whether a shape is selected, check for the shape's index in `selectionIndexes`. The selection management methods for the following `MYShapeEditorDocument` class are implemented as follows:

```
// Selection Management
- (BOOL)setControllerSelectionIndexes:(NSIndexSet *)indexes
{
  [self controllerDidBeginEditing];
  [indexes retain];
  [selectionIndexes release];
  selectionIndexes = indexes;

  [self controllerDidEndEditing];

  return YES;
}
```

```
- (NSIndexSet *)controllerSelectionIndexes
{
  if(nil == selectionIndexes)
  { // Set initially empty selection
    [self setControllerSelectionIndexes:[NSIndexSet indexSet]];
  }

  return selectionIndexes;
}


- (BOOL)controllerAddSelectionIndexes:(NSIndexSet *)indexes
{
  NSMutableIndexSet  *newIndexSet =
      [[self controllerSelectionIndexes] mutableCopy];

  [newIndexSet addIndexes:indexes];
  [self setControllerSelectionIndexes:newIndexSet];

  return YES;
}


- (BOOL)controllerRemoveSelectionIndexes:(NSIndexSet *)indexes
{
  NSMutableIndexSet  *newIndexSet =
      [[self controllerSelectionIndexes] mutableCopy];

  [newIndexSet removeIndexes:indexes];
  [self setControllerSelectionIndexes:newIndexSet];

  return YES;
}


- (NSArray *)selectedObjects
{
  return [[self shapesInOrderBackToFront] objectsAtIndexes:
      [self controllerSelectionIndexes]];
}
```

All changes to the set of selection indexes are funneled through the `-setShapeSelectionIndexes:` method, which calls `[self controllerDidBeginEditing]` before updating the selection and `[self controllerDidEndEditing]` after the update. As a result, changes to the selection cause refresh of both the custom view and the table view. A selection change made from one view is automatically reflected in the other.

When the user changes the selection in the table view, `NSTableView` informs its delegate and gives the delegate a chance to affect the change via the `-tableView:selectionIndexesForProposedSelection:` method. In addition to acting as the data source for the table view, each `MYShapeEditorDocument` instance also acts as the delegate for its table view. The following `MYShapeEditorDocument` implementation of `-tableView:selectionIndexesForProposedSelection:` keeps the controller's selection up to date.

```
// NSTableView delegate methods
- (NSIndexSet *)tableView:(NSTableView *)tableView
    selectionIndexesForProposedSelection:
    (NSIndexSet *)proposedSelectionIndexes
{
  [self setControllerSelectionIndexes:proposedSelectionIndexes];

  return proposedSelectionIndexes;
}
```

Mediating Controller Tasks (Adding and Removing Model Objects)

Adding new shape instances to the model and later removing selected shapes are best performed by Action methods (see Chapter 17, "Outlets, Targets, and Actions"). Add the following two method declarations to the interface for the `MYShapeEditorDocument` class:

```
// Actions
- (IBAction)addShape:(id)sender;
- (IBAction)removeSelectedShapes:(id)sender;
```

The Action methods are called by buttons in the View subsystem. Implement the Action methods as follows:

```
- (IBAction)addShape:(id)sender;
{
  [self controllerDidBeginEditing];
  [self setShapesInOrderBackToFront:[shapesInOrderBackToFront
      arrayByAddingObject:[[[MYShape alloc] init] autorelease]]];

  [self controllerDidEndEditing];
}


- (IBAction)removeSelectedShapes:(id)sender;
{
  [self controllerDidBeginEditing];
  NSRange    allShapesRange = NSMakeRange(0,
      [[self shapesInOrderBackToFront] count]);
  NSMutableIndexSet  *indexesToKeep = [NSMutableIndexSet
      indexSetWithIndexesInRange:allShapesRange];
```

```
  [indexesToKeep removeIndexes:[self controllerSelectionIndexes]];
  [self setShapesInOrderBackToFront:[[self shapesInOrderBackToFront]
      objectsAtIndexes:indexesToKeep]];
  [self setControllerSelectionIndexes:[NSIndexSet indexSet]];

  [self controllerDidEndEditing];
}
```

The next step is to add graphical selection and editing of shapes to the application.

Extending the `MYShapeDraw` View Subsystem for Editing

Create a subclass of `MYShapeView` called `MYEditorShapeView` with the following declaration:

```
@interface MYEditorShapeView : MYShapeView
{
  NSPoint          dragStartPoint;
}

@end
```

The `dragStartPoint` instance variable is just an implementation detail that supports graphical dragging to reposition shapes with the mouse. The partial implementation of `MYEditorShapeView` that follows is provided to show how the custom view uses its data source to implement selection and editing features, but most of the details aren't important to the Controller subsystem:

```
@implementation MYEditorShapeView


// Overrides the inherited implementation to first draw the shapes and
// then draw any selection indications
- (void)drawRect:(NSRect)aRect
{
  [super drawRect:aRect];

  [NSBezierPath setDefaultLineWidth:MYSelectionIndicatorWidth];
  [[NSColor selectedControlColor] set];

  // Draw selection indication around each selected shape
  for(MYShape *currentShape in [[self dataSource] selectedShapes])
  {
    [NSBezierPath strokeRect:[currentShape frame]];
  }
}

// Select or deselect shapes when the mouse button is pressed.
// Standard management for multiple selection is provided.  A mouse
```

```
// down without modifier key deselects all previously selected shapes
// and selects the shape if any under the mouse.  If the Shift modifier
// is used and there is a shape under the mouse, toggle the selection
// of the shape under the mouse without affecting the selection status
// of other shapes.
- (void)mouseDown:(NSEvent *)anEvent
{
  NSPoint    location = [self convertPoint:[anEvent locationInWindow]
      fromView:nil];

  // Set the drag start location in case the event starts a drag
  // operation
  [self setDragStartPoint:location];

  // ... The rest of the implementation omitted for brevity ...
}


// Drag repositions any selected shapes
- (void)mouseDragged:(NSEvent *)anEvent
{
  [[self dataSource] controllerDidBeginEditing];
  NSPoint      location = [self convertPoint:
      [anEvent locationInWindow] fromView:nil];

  NSPoint      startPoint = [self dragStartPoint];
  float        deltaX = location.x - startPoint.x;
  float        deltaY = location.y - startPoint.y;

  for(MYShape *currentShape in [[self dataSource] selectedShapes])
  {
    [currentShape moveByDeltaX:deltaX deltaY:deltaY];
  }

  [self setDragStartPoint:location];
  [self autoscroll:anEvent];  // scroll to keep shapes in view

  [[self dataSource] controllerDidEndEditing];
}

@end
```

Controllers are responsible for keeping views and models up to date with each other but can't fulfill that role if the model is changed behind the controller's back. Therefore, views must inform the controller about changes made to the model. The two bold lines

of code in the implementation of `MYEditorShapeView`'s `-mouseDragged:` method notify the controller when model objects are modified directly by the view.

You can inspect the full implementation of `MYEditorShapeView` and the Interface Builder `.nib` files in the `MYShapeEditor1` folder at www.CocoaDesignPatterns.com. Take a little time to explore `MYShapeEditor1` application. In spite of the fact that it has taken quite a few pages to describe how it all works, there really isn't very much code. Play with the application.

Redesigning and Generalizing the Solution

MYShapeEditor1 meets all of the example's requirements with straightforward method implementations written from scratch. It might seem like the mediation "glue" code is unique to this example. However, it's pretty common for Model subsystems to store arrays of objects. Certainly, more complex models may use more complex data structures or contain many different arrays of objects, but a class that generalizes the approach used in this example to mediate between any array of arbitrary model objects and multiple views can be reused in a wide variety of applications. So the challenge now is to find and encapsulate the reusable parts of this example to provide that general solution.

Start by creating a new class to implement the general solution and call that class `MYMediatingController`. Then examine the current implementation of `MYShapeEditorDocument` and identify features to move to the new class. A general mediating controller must be able to add and remove model objects, so move the `-add:` and `-remove:` Action methods to the new class. Selection management is needed in the new class, so move the `selectionIndexes` instance variable from the `MYShapeEditorDocument` to the `MYMediatingController` class. Move all of the selection management methods like `- controllerSetSelectionIndexes:` and `-controllerAddSelectionIndexes:` to the new class. Finally, a mediator for an arbitrary array of model objects needs to provide access to that array. Add a method called `-arrangedObjects` that returns an `NSArray` pointer. The `MYMediatingController` declaration should look like the following:

```
@interface MYMediatingController : NSObject
{
  NSIndexSet            *selectionIndexes;         // The selection
}

// arranged content
- (NSArray *)arrangedObjects;

// Actions
- (IBAction)add:(id)sender;
- (IBAction)remove:(id)sender;

// Selection Management
- (BOOL)controllerSetSelectionIndexes:(NSIndexSet *)indexes;
```

```
- (NSIndexSet *)controllerSelectionIndexes;
- (BOOL)controllerAddSelectionIndexes:(NSIndexSet *)indexes;
- (BOOL)controllerRemoveSelectionIndexes:(NSIndexSet *)indexes;
- (NSArray *)selectedObjects;

@end
```

After the redesign, all that's left in the `MYShapeEditorDocument` interface is the following:

```
@interface MYShapeEditorDocument : NSDocument
{
  NSArray              *shapesInOrderBackToFront; // The model
  IBOutlet NSView      *shapeGraphicView;
  IBOutlet NSTableView *shapeTableView;
}

@property (readonly, copy) NSArray *shapesInOrderBackToFront;
@property (readwrite, retain) NSView *shapeGraphicView;
@property (readwrite, retain) NSTableView *shapeTableView;

@end
```

As the coordinating controller, `MYShapeEditorDocument` needs a way to configure the mediating controller. Add an outlet called `mediatingController` to the interface of `MYShapeEditorDocument` so that document instances can be connected to a mediating controller via Interface Builder. `MYShapeEditorDocument` also needs a way to be notified when the model is changed via the Controller subsystem, so add a `-mediatingControllerDidDetectChange:` method to `MYShapeEditorDocument`. The `MYShapeEditorDocument` class is now declared as follows:

```
@interface MYShapeEditorDocument : NSDocument
{
  NSArray              *shapesInOrderBackToFront; // The model
  IBOutlet NSView      *shapeGraphicView;
  IBOutlet NSTableView *shapeTableView;
  IBOutlet MYMediatingController *mediatingController;
}

@property (readonly, copy) NSArray *shapesInOrderBackToFront;
@property (readwrite, retain) NSView *shapeGraphicView;
@property (readwrite, retain) NSTableView *shapeTableView;
@property (readwrite, retain) MYMediatingController
    *mediatingController;

- (void)mediatingControllerDidDetectChange:
    (NSNotification *)aNotification;
```

```
@end
```

Implement `MYShapeEditorDocument`'s `-mediatingControllerDidDetectChange:` to synchronize the custom shape view and the table view with the model:

```
- (void)mediatingControllerDidDetectChange:
    (NSNotification *)aNotification;
{
  [[self shapeGraphicView] setNeedsDisplay:YES];
  [[self shapeTableView] reloadData];
  [[self shapeTableView] selectRowIndexes:
      [[self mediatingController] controllerSelectionIndexes]
      byExtendingSelection:NO];
}
```

The `MYShapeEditor2` folder at www.CocoaDesignPatterns.com contains an Xcode project with the redesign completed. There is an instance of `MYMediatingController` in the document `.nib`, and the `dataSource` outlets of view objects are connected to the mediating controller. The new design is illustrated in Figure 29.4.

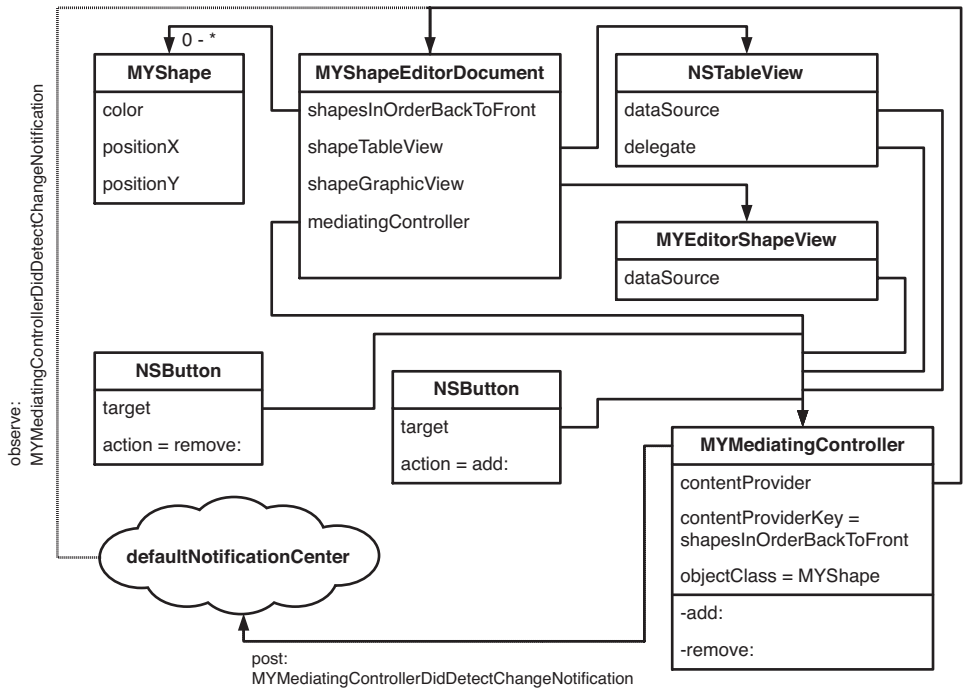

Figure 29.4 The new design of `MYShapeEditor`

The implementation MYMediatingController shouldn't have any dependencies on other classes in MYShapeEditor, or it won't be reusable in other applications. For example, when MYMediatingController adds new objects to the model, what kind of objects should it add? The class of added objects must be configurable at runtime to keep MYMediatingController general. MYMediatingController also needs a general way to get access to the array of model objects. Add the following instance variable declarations to MYMediatingController:

```
Class               objectClass;         // Class of model objects
IBOutlet id         contentProvider;     // Provider of model array
NSString            *contentProviderKey; // The array property name
```

At runtime, the contentProvider outlet is connected to whatever application-specific object provides the array of model objects. The contentProviderKey variable contains the name of the array property provided by contentProvider. Setting both the provider and the name of the provider's array property at runtime ensures maximum flexibility.

All that remains is to implement the MYMediatingController without any application-specific dependencies. The implementation of selection management and table view delegate methods are the same in MYMediatingController as they were in MYShapeEditorDocument. The rest of the code in MYMediatingController is similar to the code previously implemented in MYShapeEditorDocument, but the new code can be reused in any application. The following implementation of MYMediatingController shows the changes from MYShapeEditorDocument in bold but omits the implementations of methods that are identical in both classes to keep the listing short.

```
@implementation MYMediatingController

@synthesize objectClass;
@synthesize contentProvider;
@synthesize contentProviderKey;

- (void)dealloc
{
  [self controllerSetSelectionIndexes:nil];
  [self setContentProvider:nil];
  [self setContentProviderKey:nil];
  [super dealloc];
}


// arranged content
- (NSArray *)arrangedObjects
{
  return [[self contentProvider] valueForKey:
      [self contentProviderKey]];
}
```

```
// Actions
- (IBAction)add:(id)sender;
{
  [self controllerDidBeginEditing];
  NSArray  *newContent = [[self arrangedObjects] arrayByAddingObject:
      [[[[self objectClass] alloc] init] autorelease]];

  [[self contentProvider] setValue:newContent forKey:
      [self contentProviderKey]];

  [self controllerDidEndEditing];
}


- (IBAction)remove:(id)sender;
{
  [self controllerDidBeginEditing];
  NSRange    allObjectsRange = NSMakeRange(0,
      [[self arrangedObjects] count]);
  NSMutableIndexSet  *indexesToKeep =
      [NSMutableIndexSet indexSetWithIndexesInRange:allObjectsRange];

  [indexesToKeep removeIndexes:[self controllerSelectionIndexes]];
  NSArray  *newContent = [[self arrangedObjects]
      objectsAtIndexes:indexesToKeep];

  [[self contentProvider] setValue:newContent forKey:
      [self contentProviderKey]];

  [self controllerSetSelectionIndexes:[NSIndexSet indexSet]];

  [self controllerDidEndEditing];
}


// Editing
- (void)controllerDidEndEditing
{
  [[NSNotificationCenter defaultCenter]
      postNotificationName:MYMediatingControllerContentDidChange
      object:self];
}
```

```
// NSTableView data source methods
- (int)numberOfRowsInTableView:(NSTableView *)aTableView
{
  return [[self arrangedObjects] count];
}


- (id)tableView:(NSTableView *)aTableView
   objectValueForTableColumn:(NSTableColumn *)aTableColumn
   row:(int)rowIndex
{
  id shape = [[self arrangedObjects] objectAtIndex:rowIndex];

  return [shape valueForKey:[aTableColumn identifier]];
}


- (void)tableView:(NSTableView *)aTableView setObjectValue:(id)anObject
   forTableColumn:(NSTableColumn *)aTableColumn row:(NSInteger)rowIndex
{
  [self controllerDidBeginEditing];
  id shape = [[self arrangedObjects] objectAtIndex:rowIndex];

  [shape setValue:anObject forKey:[aTableColumn identifier]];

  [self controllerDidEndEditing];
}

@end
```

Examples in Cocoa

Compare the `MYMediatingController` class developed in the "Solution" section of this chapter to Cocoa's `NSController` and `NSArrayController` classes documented at http://developer.apple.com/documentation/Cocoa/Reference/ApplicationKit/Classes/NSArrayController_Class/Reference/Reference.html. The example in the Solution section re-invents the `NSArrayController` class and reveals both why the `NSArrayController` class exists and how it can be used in your applications.

`NSArrayController` mediates between arrays of model objects and your application's view objects; it also keeps track of selection and provides methods to add and remove model objects.

`NSTreeController` is similar to `NSArrayController` but enables you add, remove, and manage model objects in a tree data structure. `NSTreeController` is used with `NSOutlineViews`.

`NSObjectController` mediates between a single model object and your application's view objects. `NSObjectController` is the superclass of `NSArrayController` and `NSTreeController`. `NSObjectController` provides the concept of a single selected object.

`NSUserDefaultsController` encapsulates reusable code for mediating between user preferences (the User Defaults system) and your application's views.

Controllers and Bindings

Cocoa's `NSArrayController` class is more or less a drop-in replacement for the developed `MYMediatingController` class. However, `NSArrayController` uses a design that even further reduces the amount of application-specific code needed in MYShapeEditor. As shown in Figure 29.5, `MYMediatingController` posts a notification that's observed by `MYShapeEditorDocument` so that `MYShapeEditorDocument` can keep the views synchronized with the model. Cocoa provides a technology called "bindings" that provides an alternative technique for keeping objects synchronized.

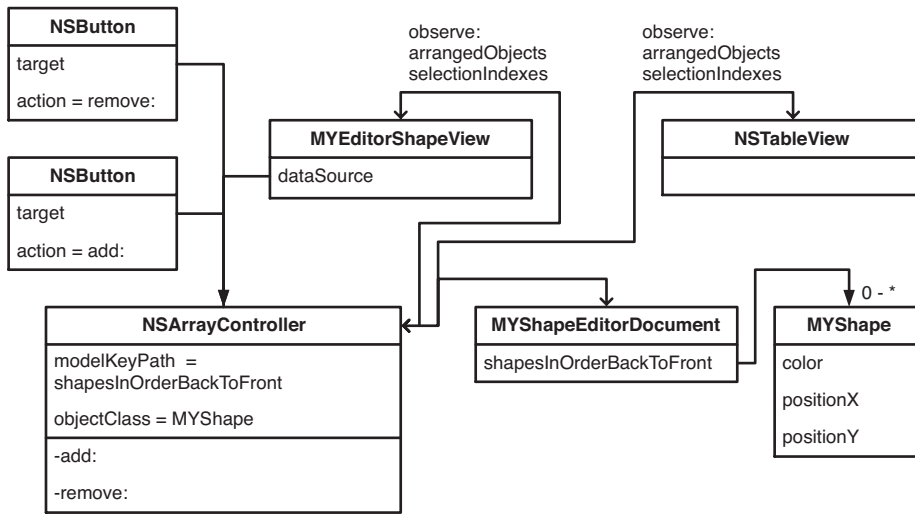

Figure 29.5 `MYShapeEditor` using `NSArrayController`

The `MYShapeEditor3` folder at www.CocoaDesignPatterns.com implements the `MYShapeEditor` application with bindings and `NSArrayController` using the design shown in Figure 29.5. The `MYShapeEditorDocument` class is simplified in `MYShapeEditor3` by removing all of the coordinating code previously used to synchronize views with the model. Bindings configured in Interface Builder replace that functionality.

Bindings are a large enough topic that they deserve their own chapter. Chapter 32, "Bindings and Controllers," describes the Bindings design pattern and its underlying implementation using lower level Cocoa design patterns.

Consequences

Cocoa's `NSController` subclasses mediate between models and views. The `MYShapeEditor` example in this chapter identifies some of code that the `NSArrayController` class replaces in typical applications, but even more controller code can be removed by using Cocoa's `NSController` subclasses together. Consider the common Master-Detail style of user interface. There is a master list of objects that can be inspected. When one of the objects is selected, details about the selected object are displayed. But what happens when the selected object is complex itself? The details for the selected object might include another list of subobjects used by the selected object. One convenient solution is to chain multiple `NSArrayController` instances together. The View that displays the selected object's list of subobjects might access the arranged objects of an array controller that that in turn accesses the selected object of another array controller, as shown in Figure 29.6.

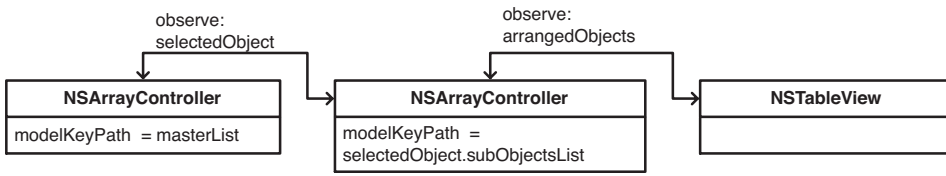

Figure 29.6 Mediating controllers are chained together to control complex relationships.

The pattern of chaining mediating controllers together highlights another reason that it is best to store selection information in the Controller subsystem instead of views. The `MYShapeEditor` example synchronizes selection between two views, but the example can be modified to enable separate selection in the two views simply by using two separate array controllers that both mediate access to the same array of model objects. The MYShapeEditor4 folder at www.CocoaDesignPatterns.com implements the separate selection design shown in Figure 29.7.

Cocoa's `NSController` subclasses reduce the amount of code needed to implement Controller subsystems and incorporate a very flexible design. Managing selection information within the Controller subsystem enables controller chaining and even a few other features that haven't been mentioned yet. For example, a button used to remove currently selected objects from the model should probably be disabled if there are no objects selected. `NSArrayController` already knows about the selection and even provides a `canRemove` property suitable for "binding" to a button's `isEnabled` property.

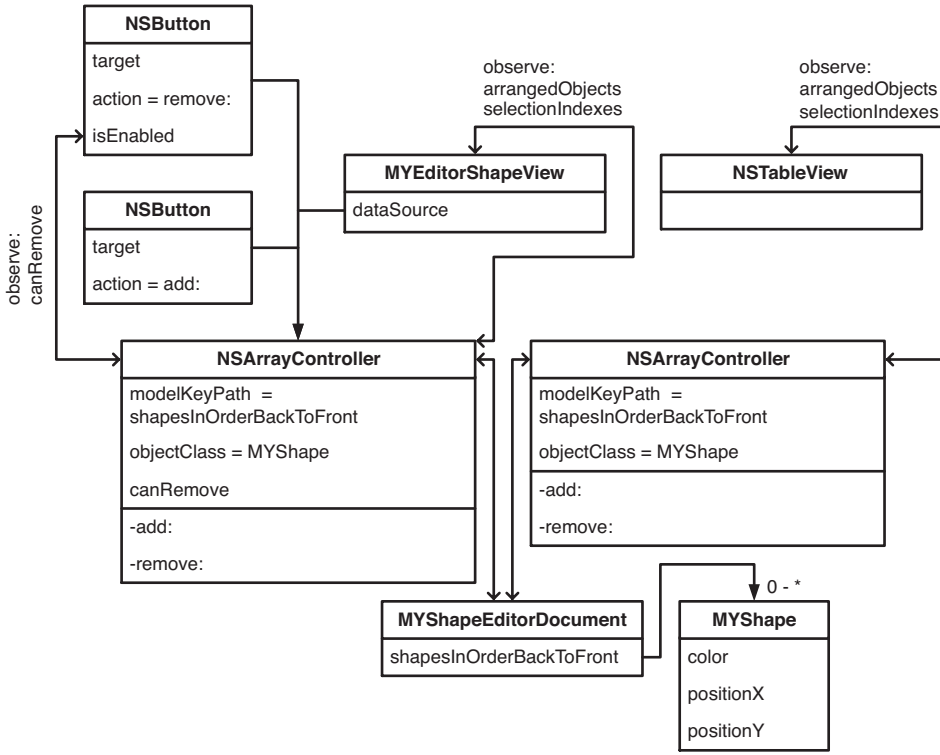

Figure 29.7 Separate `NSArrayController instances` enable separate selections in the same model.

32

Bindings and Controllers

Chapter 29, "Controllers," describes the roles of Coordinating Controllers and Mediating Controllers within Model View Controller design pattern that permeates Cocoa. Coordinating Controllers initialize, load, and save the Model and View subsystems. Mediating Controllers manage the flow of data between view objects and model objects to minimize coupling between the subsystems. Cocoa supplies the `NSApplication`, `NSDocumentController`, `NSDocument`, `NSWindowController`, and `NSViewController` classes among others to provide reusable implementations of most common coordinating tasks. Cocoa also includes `NSObjectController`, `NSArrayController`, `NSTreeController`, and `NSUserDefaultsController`, which provide reusable implementations of some common mediating tasks.

Cocoa's reusable Controller subsystem classes go a long way toward simplifying the design and development of traditional "glue" code needed to meld a model and a view into a cohesive application. The `MYShapeDraw` example in Chapter 29 shows how patterns like Outlets, Targets and Actions, Notifications, and Data Sources are used in combination with the Controllers pattern to implement full-featured Controller subsystems. However, starting with Mac OS X version 10.3, Cocoa Bindings technology has enabled a higher level of abstraction for Mediating Controllers. Bindings further reduce the amount of code needed to implement Controller subsystems and can be configured in Interface Builder to nearly eliminate code for mediating tasks.

Role of Bindings and Controllers

Bindings and Controllers work side-by-side with other patterns like Targets and Actions, Data Sources, and Notifications. You can use Bindings to reduce the amount of mediating "glue" code in your applications, but as always, there is a trade-off. Look at each application design situation on a case-by-case basis to decide which approach makes the most sense. This chapter provides the information you'll need to evaluate whether to use Bindings and Controllers or other patterns or some mixture.

Bindings keep model objects and view objects synchronized so that changes in one subsystem are automatically reflected in the other. Like almost all Cocoa technology,

bindings are implemented to reduce or eliminate coupling between objects. Bindings are based on the string names of object properties as opposed to compiled addresses or offsets, and bindings are configurable at design time and runtime.

`NSController` classes are valuable components of any Cocoa application that uses the Model View Controller pattern, whether bindings are used. In contrast, bindings should only be used in combination with controller objects like `NSObjectController` and `NSArrayController`. Whenever two objects are bound, at least one of them should be a controller. Controllers can be bound to each other. View objects can be bound to a controller. Model objects can be bound to a controller. Avoid binding View subsystem objects directly to Model subsystem objects. Don't bind view objects together or model objects together.

> **Note**
>
> There is nothing in the bindings technology that prevents direct binding from View subsystem objects to Model subsystem objects or binding View objects to other view objects or binding model objects together. However, direct bindings without the intervention of a controller are an anti-pattern as explained in "The Importance of Using Controllers with Bindings" section of this chapter.

The simplest example of binding within a Model View Controller application is shown in Figure 32.1, which depicts a text field with has its own `floatValue` property bound to the `floatValue` property of whatever object is selected by an instance of `NSObjectController`. Chapter 29 explains the concept of selection within controllers. The `NSObjectController`'s content outlet is set to an instance of `MYModel`, which provides a `floatValue` property. The content of an `NSObjectController` instance is just one object unlike an `NSArrayController` which uses an array of objects as its content. The selection provided by an `NSObjectController` is always the content object.

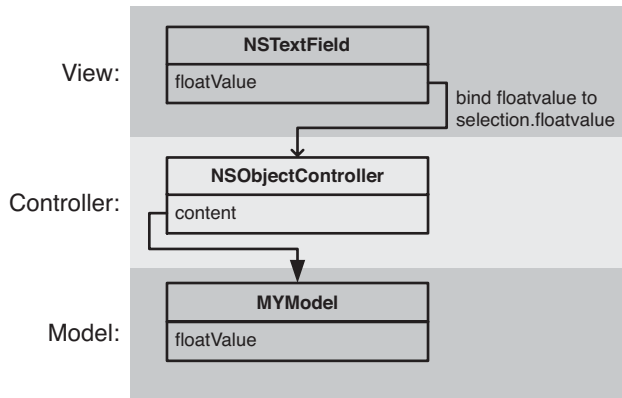

Figure 32.1 Binding within a Model View Controller application

The binding shown in Figure 32.1 keeps the `floatValue` of the text field synchronized with the `floatValue` of the `MYModel` instance. If the value in the text field is changed by the user, the change is reflected in the bound `MYModel` instance. Just as importantly, if the value in the bound `MYModel` instance is changed, the bound text field is automatically updated.

A slightly more complex binding is shown in Figure 32.2. Both a text field and a slider are bound to the `floatValue` property of a `MYModel` instance. If the user moves the slider, the `floatValue` of the `MYModel` instance is updated, which in turn causes the text field to be updated. If the user enters a value in the text field, the `floatValue` of the `MYModel` instance is updated, which in turn causes the slider to be updated. If the `floatValue` of the `MYModel` instance is changed programmatically through an appropriate Accessor method, both the slider and the text field are automatically updated to display the new value.

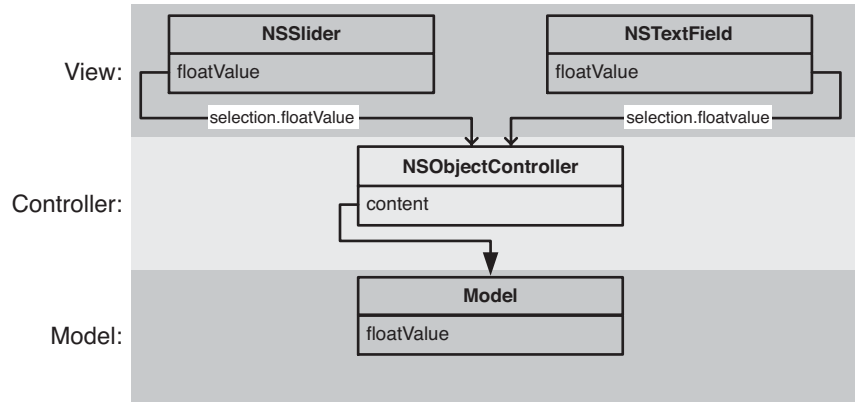

Figure 32.2 More binding within a Model View Controller application

Bindings are used in much more elaborate ways than shown in Figure 32.1 and Figure 32.2. The value of bindings is magnified when you have more complex models and more complex views. Core Data models, complex views, and the `NSController` classes integrate well with bindings and provide opportunities to almost eliminate traditional controller glue code. Nevertheless, the bindings technology is not dependent on Core Data or complex views, and all of the Cocoa technologies can be used without bindings.

Bindings Avoid Coupling

Bindings are defined by string keys that identify the objects and properties to bind. Key Value Coding (described in Chapter 19, "Associative Storage") provides the underlying mechanism used to obtain the runtime values of properties from associated keys. The use of string keys avoids any need for the objects that are bound together to know anything about each other. Any two properties of any two objects can be bound together, and as

long as properties corresponding to the string keys can be found at runtime, the binding will function. String keys minimize coupling between bound objects and allow dynamic changes at runtime. For example, if you bind a text field's value to a property of an array controller's selection, the text field will be automatically updated any time the selection changes or the value of the selected object's bound property changes. In other words, the text field isn't bound to any specific object. It's bound to whatever object is selected by the controller at any particular moment.

> **Note**
>
> Many bindings provide optional placeholder values. For example, when an object's property is bound to the selection of an array controller, it's possible to specify a placeholder value to use when there is no selection and another placeholder to use when there is multiple selection.

String keys provide even more flexibility by supporting key paths. A key path is a series of `'.'` separated keys that specify a sequence of properties to access. For example, if each employee object has a name property and a department property, and each department object has a manager who is also an employee, you could bind the value of a text field to the "`selection.department.manager.name`" key path of an array controller. At runtime, the text field's value is then synchronized to the name of the manager of the department of the selected employee. The selection is an employee object. The binding asks for the selected employee's "department" property. It then asks for the department's "manager" property. It then asks for the manager's "name" property.

It's also possible to use operators, which provide synthetic properties. For example, if each department has an array property called "employees," you can create a binding to "`selection.department.employees.@count`". The `@count` operator returns the number of objects in the array obtained from the employees property of the department property of the selected employee. A description of the operators supported for use with Cocoa collection classes is available at http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding/Concepts/ArrayOperators.html.

The Importance of Using Controllers with Bindings

Chapter 1, "Model View Controller," made the case that application data shouldn't be stored in the user interface. Instead, the Model View Controller design pattern partitions the application and stores application in a Model that's independent of any View. If you bind the properties of two View objects directly together, you are most likely diluting the benefits of Model View Controller design pattern. In the worst case, you're right back to storing crucial application data in the user interface. Therefore, it's best to bind View objects to other objects outside the View layer.

But why not bind View objects directly to Model objects? One reason is that Cocoa's `NSController` subclasses all implement the `NSEditorRegistration` informal protocol. Informal protocols are explained in Chapter 6, "Category." The `NSEditorRegistration` protocol provides methods for view objects to inform a controller when editing is underway.

It's important for controllers to have that information so that partial edits can be validated and changes can be saved without requiring the user to explicitly commit every edit that's started. `NSControllers` keep track of which view objects have unfinished edits and can force the view objects to request completion of each edit or discard the intermediate values. For example, if a user is typing in a text field and then closes the window containing the text field, the relevant `NSControllers` update the Model with the contents of the text field. The Model update causes the document to be marked as needing to be saved and then you are offered a chance to save changes before the window closes. If you don't include a controller in each binding between a View object and a Model object, then you must replace the `NSEditorRegistration` protocol functionality, and Model objects are a poor place to implement requests for completion of edits taking place in the View. Therefore, you need a controller to mediate between the View and the Model.

> **Note**
>
> Chapter 29 contains an example class similar to `NSArrayController` to show how and why reusable controller objects work. The example includes a `MYShapeEditorDocumentEditing` informal protocol similar to `NSEditorRegistration` and shows how the protocol enables coordination of changes between Model View Controller subsystems.

Another reason to include controllers in your bindings is that `NSControllers` keep track of the current selection and sometimes provide placeholder values for bound properties. Being able to bind to the current selection as opposed to a specific object makes bindings very flexible.

Finally, spaghetti bindings are as much of a problem as spaghetti code and lead to similar maintenance hassles. The discipline of including `NSControllers` in every binding clarifies the relationships between objects and serves as visual documentation for bindings. If you inspect a controller object in Interface Builder, there is a visible list of all bindings that involve that controller object, as shown in Figure 32.3. It's straightforward to inspect the controller objects whenever you open an unfamiliar `.nib` file. If bindings exist between other objects, the only way you can find them is by inspecting each end every object in the `.nib`. Religiously including controllers in bindings is a wise design guideline and serves the same purpose as coding standards: it reduces the number of places programmers need to look to understand the system.

Collaboration of Patterns Within Bindings and Controllers

Once the behavior of binding has been explained, programmers commonly want to know how bindings work. Although bindings are an advanced topic, there's really no magic. Interface Builder sends `-(void)bind:(NSString *)binding toObject:(id)observableController withKeyPath:(NSString *)keyPath options:(NSDictionary *)options` messages when it establishes bindings, and you can send the same messages to establish bindings programmatically.

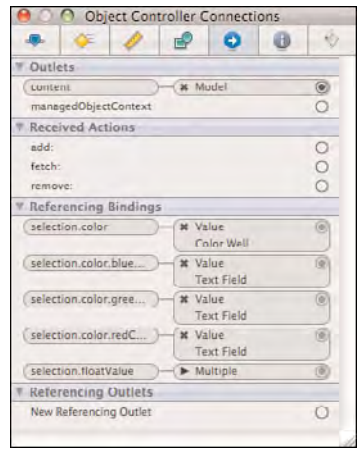

Figure 32.3 Inspecting bindings with Interface Builder

Key Value Coding and Key Value Observing technologies underlie bindings. Key Value Coding is briefly described in Chapter 19 and again in Chapter 30, "Core Data Models." It is a variation of the Associative Storage pattern, which lets you access the properties of objects as if every object were a simple dictionary of key/value pairs. See Apple's conceptual documentation for Key Value Coding at http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding/KeyValueCoding.html. Key Value Observing is a variation of the Notification pattern from Chapter 14, "Notifications." Key Value Observing monitors the values of object properties on behalf of other objects that are interested observers. The underlying implementation of Key Value Observing is somewhat different from the Notification pattern, but in essence, Key Value Observing serves the same function: Register to receive messages when something of interest happens. Apple's conceptual documentation for Key Value Observing is at http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html.

Key Value Observing is implemented by the `NSKeyValueObserving` informal protocol, which adds methods to `NSObject` from which almost all Cocoa objects inherit. Hidden deep behind the scenes, Cocoa maintains a collection of some kind that lists all of the objects that currently observe other objects' properties. Apple is deliberately vague about the specific implementation of that collection because it wants to preserve the flexibility to change the implementation in the future. You add an object to the list of objects that observe a property by calling `NSKeyValueObserving`'s `-addObserver:forKeyPath:options:context:` method. To remove an observer from the list, use `NSKeyValueObserving`'s `-removeObserver:forKeyPath:`.

What Happens in `-bind:toObject:withKeyPath:options:`?

Sending the `-bind:toObject:withKeyPath:options:` message to an object creates a bi-directional set of Key Value Observing associations. Somewhere inside Apple's `-(void)bind:(NSString *)binding toObject:(id)observableController withKeyPath:(NSString *)keyPath options:(NSDictionary *)options` implementation, the following code or something similar is executed:

```
[self addObserver:observableController forKeyPath:binding
   options:(NSKeyValueObservingOptionNew|NSKeyValueObservingOptionOld)
   context:nil];

[observableController addObserver:self forKeyPath:keyPath
   options:(NSKeyValueObservingOptionNew|NSKeyValueObservingOptionOld)
   context:nil];
```

There isn't much more involved with the establishment of bindings. Apple documents the available options at http://developer.apple.com/documentation/Cocoa/Reference/ApplicationKit/Protocols/NSKeyValueBindingCreation_Protocol/Reference/Reference.html. If a key path has multiple `'.'` separated properties, `-bind:toObject:withKeyPath:options:` adds observers for all of the individual properties in the path as needed. You can get information about existing bindings via the `-(NSDictionary *)infoForBinding:(NSString *)binding` method. Sending the `-(void)unbind:(NSString *)binding` message results in corresponding calls to `NSKeyValueObserving`'s `-(void)removeObserver:(NSObject *)anObserver forKeyPath:(NSString *)keyPath` method.

Given that bindings are a relatively thin veneer on Key Value Observing, the magic of bindings resides within Key Value Observing.

How Does Key Value Observing Detect Changes to Observed Properties so That Observing Objects Can Be Notified?

The answer is that changes to observed properties need to be bracketed by calls to `-(void)willChangeValueForKey:(NSString *)key` and `-(void)didChangeValueForKey:(NSString *)key`. If you write your own code to programmatically modify the values of observed properties, you may need to explicitly call `-willChangeValueForKey:` and `-didChangeValueForKey:` like the following method that sets the "counter" property without calling an appropriate Accessor method:

```
- (void)incrementCounterByInt:(int)anIncrement {
   [self willChangeValueForKey:@"counter"];
   counter = counter + anIncrement;
   [self didChangeValueForKey:@"counter"];
}
```

Inside `NSObject`'s default implementation of the `NSKeyValueObserving` informal protocol, `-willChangeValueForKey:` and `-didChangeValueForKey:` are implemented to send messages to registered observers before and after the property value changes.

It's not necessary to explicitly call `-willChangeValueForKey:` and `-didChangeValueForKey:` within correctly named Accessor methods. When you use Objective-C 2.0's `@synthesize` directive to generate Accessor method implementations, the details are handled for you. Even if you hand-write Accessor methods, Cocoa provides automatic support for Key Value Observing through a little bit of Objective-C runtime manipulation briefly described at http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueObserving/Concepts/KVOImplementation.html. At runtime, Cocoa is able to replace your implementation of each Accessor method with a version that first calls `-willChangeValueForKey:`, then calls your implementation, and finally calls `-didChangeValueForKey:`.

When Key Value Coding's `-(void)setValue:(id)value forKey:(NSString *)key` or `-(void)setValue:(id)value forKeyPath:(NSString *)keyPath` methods are used to modify an observed property, the appropriate Accessor methods (if any) are called, and the Accessor methods take care of calling `-willChangeValueForKey:` and `-didChangeValueForKey:`. If there aren't any available Accessor methods, `-setValue:forKey:` and `-setValue:forKeyPath:` call `-willChangeValueForKey:` and `-didChangeValueForKey:` directly. In summary, you only need to explicitly call `-willChangeValueForKey:` and `-didChangeValueForKey:` if you change the value of an observed property without using Key Value Coding and without using an appropriately named Accessor method.

> **Note**
>
> As recommended in Chapter 10, "Accessors," if you consistently use Accessor methods to access or mutate properties, you will save yourself a lot of work. In addition to the memory management advantages of using accessors, you'll also avoid the need to ever explicitly call `-willChangeValueForKey:` and `-didChangeValueForKey:`.

What Message Is Sent to Notify Registered Observers When an Observed Property's Value Is Changed?

By default, the `-didChangeValueForKey:` method sends the `-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context` message to all registered observers after an observed property changes value. You can configure the `-willChangeValueForKey:` method to send notification before each change if you specify the `NSKeyValueObservingOptionPrior` option in the `options:` argument used to register an observer. The `options:` argument also governs whether the change notification includes only the previous value, only the new value, or both the old and new values.

Most Cocoa View subsystem classes already implement `-observeValueForKeyPath:ofObject:change:context:`. You need to implement that method in your custom View objects if you want them to work correctly with bindings. You may also need to implement `-observeValueForKeyPath:ofObject:change:context:` in model objects if you want to perform special logic whenever observed properties change. Unfortunately, implementing `-observeValueForKeyPath:ofObject:change:context:` is one of the least elegant aspects of using Cocoa.

> **Note**
>
> You are able to specify an Objective-C selector that identifies the message you want to receive when you use Cocoa's `NSNotificationCenter`. Selectors are explained in Chapter 9, "Perform Selector and Delayed Perform," and `NSNotificationCenter` is explained in Chapter 14. In contrast Key Value Observing always notifies observers via the `-observeValueForKeyPath:ofObject:change:context:` method.

You almost invariably have to implement `-observeValueForKeyPath:ofObject:change:context:` by using string comparisons to determine what logic to invoke based on which key path changed. The following code is a trivial example implementation of `-observeValueForKeyPath:ofObject:change:context:`:

```
- (void)observeValueForKeyPath:(NSString *)keyPath
   ofObject:(id)object change:(NSDictionary *)change
   context:(void *)context
{
   if ([keyPath isEqualToString:@"floatValue"]) {
      NSNumber   *newValue = [change
         objectForKey:NSKeyValueChangeNewKey];
      if(0.0 > [newValue floatValue]) {
         // Perform special logic for negative values here
      }
      [self setNeedsDisplay:YES];
   }

   // be sure to call the super implementation
   [super observeValueForKeyPath:keyPath
      ofObject:object change:change
      context:context];
}
```

The need to perform explicit string comparisons like `[keyPath isEqualToString:@"floatValue"]` in `-observeValueForKeyPath:ofObject:change:context:` is inelegant. It's easy to imagine an implementation of `-observeValueForKeyPath:ofObject:change:context:` that has to perform hundreds of string comparisons after every observed property change to control application logic. Objective-C selectors and the Perform Selector pattern from Chapter 9 exist to make string comparisons in branch logic unnecessary. It's unfortunate that Apple didn't take advantage of the pre-existing patterns like `NSNotification` and the use of selectors when implementing Key Value Observing.

> **Note**
>
> The Associative Storage pattern is a prominent building block of Key Value Coding, Key Value Observing, and Bindings. Dictionaries containing key/value pairs specify the options to binding methods. A dictionary provides information about changes in

> `-observeValueForKeyPath:ofObject:change:context:`. And Key Value Coding is itself a variation of the Associative Storage pattern.

One way to make `-observeValueForKeyPath:ofObject:change:context:` a little bit more elegant is to use key path strings as notification names as follows:

```
- (void)observeValueForKeyPath:(NSString *)keyPath
   ofObject:(id)object change:(NSDictionary *)changeDictionary
   context:(void *)context
{
   // copy the change dictionary and add the context to it
   NSMutableDictionary  *infoDictionary = [NSMutableDictionary
      dictionaryWithDictionary:changeDictionary];
   [infoDictionary setObject:context forKey:@"MYBindingContext"];

   // post a notification to interested observers using the key path as
   // the notification name
   [[NSNotificationCenter defaultCenter] postNotificationName:keyPath
      object:object userInfo:infoDictionary];

   // be sure to call the super implementation
   [super observeValueForKeyPath:keyPath
      ofObject:object change:change
      context:context];
}
```

If you use the approach of converting Key Value Observation notifications into `NSNotifications`, you can have any number of observers that each register a different selector for the same key path. Unfortunately, the `NSNotification` approach has problems of its own. Using key path strings as notification names is not ideal because key paths are specified in Interface Builder and must be duplicated exactly in your code that registers for notifications. A simple change in Interface Builder could necessitate changes to notification code in multiple disparate places within your application. The compiler can't detect errors in the key path strings, so you must test at runtime to detect key path errors. Nevertheless, `NSNotificationCenter` provides at least one way to circumvent the use of explicit string comparisons in your own code.

Bindings and Controllers Limitations and Benefits

A common criticism of bindings is that there is too much magic happening that the programmer can't see. This chapter dispels some of the magic. Bindings are hard to document because they typically aren't visible in code. The same criticism can be made for Targets, Actions, and Outlets that are configured in Interface Builder. However, due in part to the flexibility and potential complexity of bindings, the need to document bindings is even greater than the need to document Targets, Actions, and Outlets.

The use of string keys avoids coupling between objects. Any two properties of any two objects can bind together as long as properties corresponding to the string keys can be found at runtime. Of course, the corresponding down side is that the compiler can't determine correctness of bindings. You have to wait until runtime to test bindings.

Bindings interoperate with features like Value Transformers that aren't covered in this chapter (see http://developer.apple.com/documentation/Cocoa/Conceptual/ValueTransformers/Concepts/TransformersAvail.html). Bindings have the potential to replace code that would otherwise need to be written. Chapter 29 culminated with an example use of Bindings. That example highlights the code that's replaced when bindings are used.

A

B

C

D

E

F

G

H

I

J–K

L–M

N

O

P

Q–R

S

T

U

V

W

X–Z