The Pragmatic Programmers

Programming Cocoa with Ruby

Create Compelling Mac Apps Using RubyCocoa

Brian Marick

Edited by Daniel H Steinberg

The Facets of Ruby Series

What Readers Are Saying About *Programming Cocoa with Ruby*

This isn't just a book on RubyCocoa; it is probably the best book I've seen that explains Cocoa technology. It actually explains how some of the core technologies, especially bindings, work instead of just showing an example of how to use them.

► **Allison Newman**
Cocoa application developer

Learning a new API is hard enough, but learning a new API and a new programming language at the same time can be overwhelming. *Programming Cocoa with Ruby* is written for those of us used to a language like Ruby or Python who want to learn about all the great stuff Cocoa has to offer.

► **Jeremy McAnally**
Developer, entp

The influence of Smalltalk on Ruby and Objective-C is considerable. It shouldn't be a surprise then that Cocoa, whose native tongue is Objective-C, can be effectively learned and programmed from Ruby in a way that captures the succinctness and expressiveness of this newly popular scripting language. Brian's book is a great introduction to the agile development of Cocoa apps, it serves as a primer on Cocoa, and it demonstrates sound and thoughtful development practices and hygiene throughout.

► **Jerry Kuch**
Principal engineer, EMC Corporation

Programming Cocoa with Ruby

Create Compelling Mac Apps Using RubyCocoa

Brian Marick

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-19-0

ISBN-13: 978-1-934356-19-7

Printed on acid-free paper.

P1.0 printing, July 2009

Version: 2009-8-6

Contents

Chapter 1

Introduction

It's simple, really: if you like Ruby and you like Macs and you want to put the two together, this book is for you. After you read through it. . . wait, scratch that—after you *work* through it, you'll be able to build nice, Mac-like apps. You'll have the memory of doing, at least once, many of the tasks that make up building a Mac app. Your life will have much less of the "What do I do *now*?" frustration that sinks so many first attempts to use a big and complicated framework.

I endured that frustration for you. When I started writing the book, I knew practically nothing about coding Mac GUI apps (in Ruby or any other language). I learned how in my usual way: by diving into coding something too ambitious. As always happens, I spent much of my time blundering down blind alleys, staring at app crashes and weird behaviors, figuring out what pieces of the conceptual puzzle I was missing, searching for them in vast masses of documentation, and revisiting old code in the light of new understanding. The only difference was that after I figured something out, I wrote a new chapter about what I'd done and what I'd learned—except that I removed most of the frustration from the story line. When I had to backtrack because I didn't know something, I wrote that something into the story just when I would have needed it. The result is what Imre Lakatos, the philosopher of science, called a *rational reconstruction* of history: follow the book, and I think you'll get pretty much the experience I *should* have had.

That turns out to be a fantastically time-consuming way of writing a book. The payoff is that, when I wrote, the experience of learning was fresh in my mind. Experts often have a problem remembering what it was like to learn and how much they used to not know. I solved that problem by being *barely not ignorant* as I wrote.

1.1 What Is Cocoa?

Most modern Mac applications are written using the Cocoa framework. Cocoa is an object-oriented framework that structures your app and handles a lot of drudgery for you. It mostly makes developing a user interface easier, but it also has classes and libraries for handling the file system, interprocess communication, persistent data, and so on.

1.2 What Is RubyCocoa?

The Cocoa framework was originally designed to be used via Objective-C programs. Objective-C is an object-oriented dialect of C. Early in the history of OS X, Apple also provided Java interfaces to some of the framework, but that didn't work out well. The problem was that Java wants to know things at compile time that Objective-C defers to run-time. For example, Objective-C is not nearly so picky about declaring types as Java is. Because Cocoa framework writers took advantage of such features, the mapping between Java and Cocoa was clumsy.

It's much easier to build a bridge between Cocoa and Ruby because Ruby and Objective-C stem from similar philosophies of language design. RubyCocoa is that bridge. With it, you can write Ruby code that calls Objective-C code, and vice versa. So, it's quite possible to write a Mac app in Ruby.

1.3 What's It Like to Learn Cocoa Using Ruby?

It doesn't take much time to learn the basics of RubyCocoa. Once you've done that, you'll spend most of your time learning Cocoa. Your Cocoa learning will occasionally be interrupted by some surprising RubyCocoa fact, which you will then absorb and move on.

This book follows that sequence. It begins with an introductory chapter that teaches RubyCocoa basics without many of the distractions of a real user interface. Then it follows a typical development pattern: begin with a small version of your app, get it working, find the next thing you wish it did, make it do that, and repeat until you're done. Cocoa and RubyCocoa topics will be covered as they come up in a realistic course of development.

With any complex framework, there's a moment when you realize you finally have a *feel* for it—when faced with a new problem, you're able to

guess whether the framework addresses it, roughly how it will address it, and where to look to find the details. The aim of this book is to give you that feel for Cocoa and RubyCocoa. It's not a reference to the Cocoa framework because that information is already on your hard drive or, at most, an HTTP address away.

Still, because no single development history can naturally encounter every topic and because exploring some topics in enough detail would be too much of a digression from the story line, the last part of the book consists of essays on important topics. They can be read in any order, at any time.

1.4 RubyCocoa? That's *So* Last Year!

During the writing of this book, various people suggested that it be switched from RubyCocoa to MacRuby, Apple's next generation of support for Ruby. As I write, though, MacRuby is still in beta. Both my limited experience with it and my subscription to the developer mailing list make me think it's not quite ready to replace RubyCocoa. As I noted in the previous section, most of your time spent learning RubyCocoa will be spent learning the Cocoa part. That's the same in RubyCocoa and MacRuby, so you might as well learn it in the more stable environment.

At some point, you'll switch to MacRuby. If you like to be on the cutting edge, you'll do it soon. If not, it will be later. In either case, it helps to know what's coming. For that reason, I've written Chapter 25, *MacRuby*, on page 334 for you.

1.5 Prerequisites

- You should have used a Mac enough that you're familiar with the conventions Mac apps follow. There's no need to have ever built an app with a graphical user interface, whether for the Mac or for any other platform. You don't need to understand Objective-C (or C).
- You should know Ruby reasonably well. A good measure of that is whether you're comfortable reading parts of someone else's Ruby code. If some gems' behavior surprises you, do you follow a stack trace into it to see what's really going on?

  I'll use some tricky Ruby code behind the scenes, but the code you'll need to understand will be fairly straightforward. However, I won't stop to explain common idioms like this sort of initialization:

  ```
  @var ||= 5
  ```

If you don't think you know Ruby well enough, I recommend the Pickaxe book, *Programming Ruby* [TFH08], possibly supplemented with *The Ruby Way* [Ful06]. My own *Everyday Scripting with Ruby* [Mar06] teaches Ruby in the same style as this book teaches RubyCocoa—by having you implement projects alongside the book—but it may be too slow-paced for an experienced programmer, and you'll still want the Pickaxe book for reference.

- Make sure you're running Apple's version of Ruby. You can confirm that like this:

```
$ /usr/bin/which ruby
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ruby
```

  If you see other output (like /usr/local/bin/ruby or /opt/local/bin/ruby), adjust your load path so that /usr/bin/ruby is used instead.

- You should be running Mac OS X 10.5 (Leopard) or later, with the Developer Tools installed.[1] Earlier versions of the Developer Tools do not include RubyCocoa.

  To see what version of RubyCocoa you're running, type this to irb:

```
irb(main):001:0> require 'osx/cocoa'
=> true
irb(main):002:0> OSX::RUBYCOCOA_VERSION
=> "0.13.1"
```

  I also have most of the examples write RubyCocoa's version to the console (visible via the Console app). If you manually installed a version of RubyCocoa before you installed Leopard, that old version may be loaded instead of Leopard's (by an application, but not by irb). If samples behave oddly, you check the console, and the version is old, then delete /Library/Frameworks/RubyCocoa.framework.

  RubyCocoa is available for earlier versions of OS X, but I've made no attempt to avoid Leopard features. Moreover, the two main Apple tools we'll use (Xcode and Interface Builder) changed considerably between 10.4 and 10.5. So if you install RubyCocoa on a pre-Leopard version, prepare to spend time figuring out how a picture in the text maps into an old tool.

- You have to be prepared to build the app yourself. If you just read the book, the knowledge probably won't stick. You'll then find that

1. You had to choose to install the Developer Tools when you installed Leopard. If you didn't, you can fetch them off the install disc or from http://developer.apple.com.

building your first app is a flood of tasks you vaguely know you should perform but forget how. It's better to build up your "muscle memory" by building the book's app before you build your own.

- You need to download bmrc-code.zip from http://pragprog.com/titles/bmrc/source_code. It contains some files and tools you'll need as you work through the chapters. When you unzip bmrc-code.zip, it will place all its contents in a code subdirectory. Since you'll likely want to rename that, I'll leave off the code prefix when referring to files. So, for example, when I direct you to the very first file you'll work on, I'll refer to it as statusbar/most-basic-app.rb.

  The download also contains snapshots of the app taken just before and after each important step. As you work along, you can copy snippets from the snapshots or use one of them as a starting point.

  The download means that, at any given moment, there may be thirty-four versions of the app on your disk: thirty-three intermediate versions that I supply and one that you're working on. That presents a problem. For example, on page 96, I write "look in WindowController to see how it handles the AppChosen notification." The problem is that your version might not use the same names as mine, so it might not handle anything called AppChosen. In that case, you'll need to look at my most recent version, not yours. But which is most recent? You can find that out by looking backward to the most recent code snippet that identifies its source file. The following snippet, for example, would tell you we're working on the reshaped-with-dsl version of the app:

  Download fenestra/reshaped-with-dsl/WindowController.rb

```
on_local_notification AppChosen do | notification |
  @logWindow.title = notification[:app_name]
end
```

1.6 Versions

The book uses these tools:

- Ruby 1.8.6
- RubyCocoa 0.13.1
- Xcode 3.0
- Interface Builder 3.0

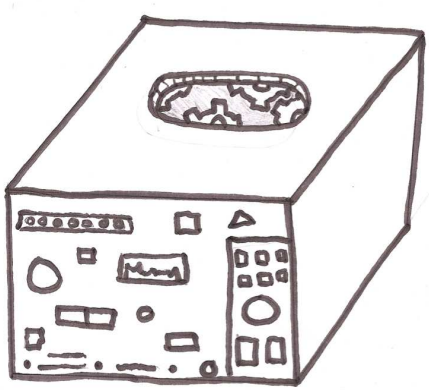


Figure 1.1: Visible workings

These were the most recent versions delivered by Apple at the time of writing.

When the book also uses libraries or gems, they're included in the book's bmrc-code.zip file.

1.7 Our Example App

If you're going to go to the trouble of working through an entire book of code, you ought to end up with something more than knowledge at the end of it. You ought to get code you can use, either as a complete app or as snippets and templates to incorporate into your own apps. I'm not creative enough to imagine an app with wide appeal, but I do know of a template you might very well need.

You see, too many programmers are like the proverbial shoemaker's children who go unshod. You build sophisticated graphical interfaces that let bond traders or camera buffs or physicists easily see and manipulate the stuff of their work, but you don't do the same for yourself. A person who one moment is adding Ajax wizardry to streamline a web app's workflow will, the next moment, be trying to diagnose a bug by groveling through textual log files, manually trying to reproduce the steps to the failure, viewing the HTML source of page after page, and poring over database tables that hold a distorted version of the app's objects. That doesn't seem right.

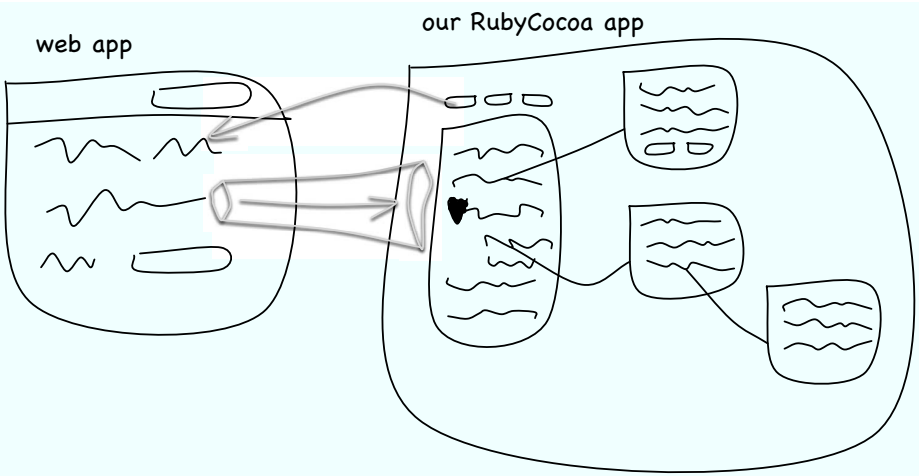


Figure 1.2: An alternate GUI

It's not right, and it's our fault. We allow our apps to be childish. Every parent knows how bad children are at answering the two vital questions: "All right, who did it?" and "What *on Earth* were you thinking to make that seem like a good idea?" Trying to figure out a bug with nothing more than a user's report and the app's regular GUI is like being faced with the child who answers every question with "I dunno." But going after a bug with general-purpose tools like the language's debugger is like talking to the child who tries to get out of trouble by throwing up a smoke screen of detail, irrelevancies, and finger-pointing. What we need is to build every app with a special window into its inner workings that programmers and testers can use (Figure 1.1, on the preceding page).

The main example in this book will be a free-standing app that peeks and pokes at another app through such a window. I'll target a web app in the book because that happens to be what I need right now, but the same principle and much of the code would apply to any app. A sketch of it in use is shown in Figure 1.2. It will contain a running log of actions the web app has taken that is expandable to the desired level of detail. A click of a button will tell the web app to undo or redo an action. Our app, helped by the web app, will know when links refer to domain objects (like user accounts), so the right gesture will pop up an understandable, tweakable, and draggable representation of the object. There's much more that such an app could do, but since this *is* a book

about programming OS X, I'll limit myself to features that teach Cocoa.

I think of such apps as windows you can both look and reach into, but the word *window* is already taken, so I'll use the Latinate equivalent: *fenestra*. It's a particularly apt choice because that's also the word for a hole surgically carved into a body part to let bad stuff leak out. In our case, the "bad stuff" is information about bugs.

The act of creating such a hole is called *fenestration*. I'll use that word when I need to describe what the program is doing, and I'll use *fenestra* to describe the result.

1.8 Centuries of the Bookmaker's Art: Scorned

I hate it when words refer to a figure that you have to flip the page to see. It's bad enough with drawings, but it really hurts my comprehension when the figure contains code. I've made a real effort to keep the code and the references to that code on the same or facing pages. The result is. . .

. . . big blank spaces at the bottom of some pages. Text and figures are traditionally laid out to avoid that ugliness. I embrace it. You deserve an unattractive book.

1.9 Some Terminology

While writing the book, I sometimes had a choice between consistent-but-awkward usage and flowing-but-inconsistent usage. For example, consider the following bit of Ruby:

```
"foo".upcase
```

I'd normally say that Ruby "sends the message :upcase to the string *"foo"*." Sometimes, though, the words *sends* and *message* won't work in a sentence, so I use "calling the method" instead. There's nothing but stylistic significance to the choice—I don't mean a different thing.

Similarly, when writing of a variable, I might say that it "refers to," "names," or "points at" an object. I might also say it "is" an object—even though that's strictly incorrect—because "i names 5" sounds silly.

1.10 Service After the Sale

My mail address is marick@exampler.com.

You can find errata at http://pragprog.com/titles/bmrc/errata.

1.11 Solving Problems

The Apple documentation (cited throughout the book) will be your main source of Cocoa information, but don't be surprised when you run into problems that it doesn't help you solve. Cocoa is *big*.

When I've been stumped, I've had the best luck just using Google to search for the right keywords. A thoroughly gratifying percentage of the time someone has written a blog entry or email message solving my exact problem. The solutions are almost always in Objective-C, but I expect that won't be a problem for you after you finish the book.

There is a low-volume mailing list, https://lists.sourceforge.net/lists/listinfo/rubycocoa-talk, where you can ask questions about RubyCocoa. The main Cocoa developer mailing list, http://lists.apple.com/mailman/listinfo/cocoa-dev, has much higher volume. If you can phrase your question

in terms that make sense to an Objective-C programmer, you can get help there. It's not unusual for Google to point me to its archive.

In /Developer/Examples, Apple provides examples of Cocoa features in the form of small—but complete—apps. They're written in Objective-C. I am ever so gradually translating them into Ruby at http://github.com/marick/cocoa-examples-translated. (I welcome help.)

When it comes to RubyCocoa itself, I've used both its source and tests to answer my questions. I encourage you to download RubyCocoa from http://rubycocoa.sourceforge.net.

1.12 Acknowledgments

Dawn, light of my life.

The creators of RubyCocoa: Eloy Duran, Fujimoto Hisa, Chris Mcgrath, Satoshi Nakagawa, Jonathan Paisley, Laurent Sansonetti, Chris Thomas, Kimura Wataru, and others.

Corey Haines, for spending two days of his Pair Programming Tour[2] in my living room, helping me figure out the mysteries of drag and drop.

My editor, Daniel Steinberg.

Technical reviewers Chris Adamson, Julio Barros, Craig Castelaz, Michael Ivey, Jerry Kuch, Mathias Meyer, Allison Newman, and Scott Schram.

Readers of the beta drafts: Steven Arnold, Jason M. Batchelor, Rune Botten, Tom Bradford, Stephyn G. W. Butcher, Leroy Campbell, Gregory Clarke, Eloy Duran, Frantz Gauthier, Joseph Grace, Aleksey Gureiev, Christopher M. Hanson, Cornelius Jaeger, Masahide Kikkawa, Frederick C. Lee, Jay Levitt, Tim Littlemore, Nick Ludlam, Stuart Malin, Ule Mette, James Mitchum, Steve Ross, Peter Schröder, Jakub Suder, Tommy Sundström, Matthew Todd, Daniel J. Wellman, Markus Werner, "Dr. Nic" Williams, and perhaps others whose names I didn't write down. (Sorry.)

Although this is a book about RubyCocoa, I've snuck in bits and pieces of a philosophy and pragmatics of application design. It's a style I have learned from people such as Kent Beck, Ward Cunningham, Carl Erickson, Michael Feathers, Martin Fowler, Steve Freeman, Richard P.

2. http://programmingtour.blogspot.com

Gabriel, Andy Hunt, Ron Jeffries, Ralph Johnson, Joshua Kerievsky, Yukihiro Matsumoto, Nat Pryce, Richard Stallman, Guy Steele, Pragmatic Dave Thomas, and others. Without them, this book would have been less interesting to write and—I hope—read.

Sophie Marick for the picture on page 16.

Giles Bowkett for the musical accompaniment to the embedded video in the sample app's help pages.

The baristas at Bar Guiliani, Aroma Cafe, and Espresso Royale, especially Alex Kunzelman for not calling the nice people from the mental health center around the fourth draft of Chapter 7.

Chapter 2

How Do We Get This Thing Started?

We're going to start fast, small, and with the fundamentals. To that end, here's the smallest RubyCocoa app:

Download statusbar/most-basic-app.rb

```
#!/usr/bin/env ruby

❶ require 'osx/cocoa'

❷ OSX::NSApplication.sharedApplication
❸ OSX::NSApp.run
```

If you run it, you'll see that it does nothing but exist: no windows, no output, no exit:

```
$ ruby most-basic-app.rb
```

(You'll have to kill the script with Control-C.)

What causes this nonbehavior? Line ❶ creates the ability for Ruby and Objective-C methods to call each other. You'll see more about how that works throughout the book.

Line ❷'s NSApplication is the class corresponding to the entire application itself. The call to class method sharedApplication creates the single instance of that class, names it with the constant NSApp (used on the next line), connects it to the window server, and does other useful initialization.

The NS prefix is used by all the Cocoa classes you're ever likely to see. So, using the module prefix OSX:: isn't really helping to avoid name clashes. I'll usually just include the OSX module and forget about it.

Notice that sharedApplication isn't an idiomatic Ruby name. In Ruby, you would be much more likely to see shared_application. It is idiomatic Objective-C, though. You should expect to see such method names—and even stranger ones—in RubyCocoa programs.

At ❸, NSApp is told to run. It does, waiting forever for someone to send it work to do.

2.1 A Program That Prints

In this section, I'm going to make the app do one tiny additional thing: print a message to standard output. That raises two questions right away:

- *When should the message be printed?* Since the Cocoa runtime tells NSApp when it (NSApp) has finished launching, that seems like a good moment.
- *Where do we add the code?* In some systems, you'd add the code in a subclass of NSApplication. That style is rare for Cocoa apps. Instead, Cocoa programmers have NSApp delegate some of the work to another object (called, unsurprisingly, a *delegate*).

The application structure we'll use looks like Figure 2.1, on the next page. Here's the code:

Download statusbar/no-ui.rb

```
#!/usr/bin/env ruby

require 'osx/cocoa'
include OSX

❶ class AppDelegate < NSObject
❷   def applicationDidFinishLaunching(aNotification)
      puts "#{aNotification.name} makes me say: Hello, world"
    end
end

❸ our_object = AppDelegate.alloc.init
  NSApplication.sharedApplication          # Creates global NSApp
❹ NSApp.setDelegate(our_object)
  NSApp.run
```

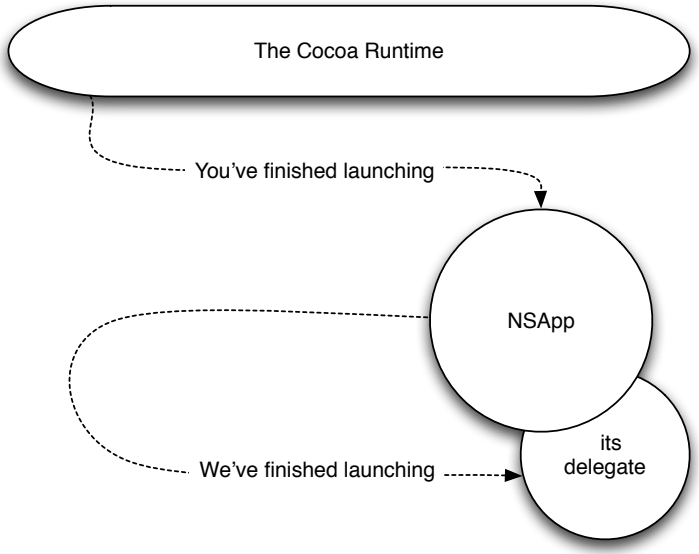


Figure 2.1: Delegating work

The delegate is defined at ❶. It volunteers to handle the "application has launched" event by defining the applicationDidFinishLaunching method (at ❷). There are other events; because the class doesn't contain a corresponding method, it won't have to handle them.

Notice what's happening here: Objective-C, like Ruby, encourages *duck typing*. The term comes from the saying "If it quacks like a duck, it's a duck." If an AppDelegate object responds to the right messages, it's a suitable application delegate no matter what its declared superclass. It does have to be an Objective-C object, so it's declared as a subclass of NSObject. That class is the root of the Objective-C class hierarchy, just as Object is the root of the Ruby hierarchy.

The app creates the delegate at ❸. Whereas in Ruby you'd expect AppDelegate.new, object creation in Objective-C is done in two parts. alloc creates the object in memory and attaches all its methods to it. init is what we in Ruby call initialize. Objective-C separates the two because many Objective-C classes have variant init methods to support different ways of initializing the instance.[1]

1. Ruby actually works like Objective-C under the covers: new first uses the class method allocate and then sends initialize to the resulting instance.

NSApp is told of its delegate at ❹. Then NSApp starts running. Once the application setup work is finished, NSApp calls its delegate's applicationDidFinishLaunching method.

There's not much more to the app: applicationDidFinishLaunching prints. For fun, I had it print the name of its single argument, an NSNotification object. Notice that sending a message to an Objective-C object is no different from sending one to a Ruby object. (You can check that aNotification names an Objective-C object by printing its class: does it start with NS?)

If you run the new script, you'll see this:

```
$ ruby no-ui.rb
NSApplicationDidFinishLaunchingNotification makes me say: Hello, world
```

Try This Yourself

1. applicationDidFinishLaunching is not the only message that can be sent to a delegate. It's too early for you to handle most of them, but try changing no-ui.rb to handle applicationWillFinishLaunching.[2]

2. Notifications have more than names. They can also point to an object of interest. In the case of these two notifications, that object is NSApp. Write some code to make sure that's true. Use the object message to get the object of interest.

3. Run your solution to the previous exercise again, but this time misspell the word *object*. Learn to recognize the output—you'll be seeing it a lot in your RubyCocoa career. That's what happens when RubyCocoa tries to send any Objective-C message that the object doesn't respond to. It's the equivalent of this Ruby error message:

```
irb(main):005:0> notification.objct
NoMethodError: undefined method `objct' for #<Notification:0x10a4050>
        from (irb):5
```

2. What's the point of knowing an application hasn't finished launching yet? Certain things happen between the two events you know about. For example, it's between them that an application is told it was started by double-clicking a file. If you want to do any setup before then, applicationWillFinishLaunching is the time to do it.

2.2 Putting an Item in the Status Bar

Without using Interface Builder (described in Chapter 3, *Working with Interface Builder and Xcode*, on page 37), making a user interface is tedious work, so we will create only a single, incredibly simple user-interface element in this section: an icon in the status bar (the symbols and text on the strip across the top of the screen on the right).

Here's the code:

Download statusbar/statusbar-item.rb

```
#!/usr/bin/env ruby

require 'osx/cocoa'
include OSX

class App < NSObject
  def applicationDidFinishLaunching(aNotification)
❶   statusbar = NSStatusBar.systemStatusBar
❷   status_item = statusbar.statusItemWithLength(NSVariableStatusItemLength)

❸   image = NSImage.alloc.initWithContentsOfFile("stretch.tiff")
❹   raise "Icon file 'stretch.tiff' is missing." unless image

❺   status_item.setImage(image)
  end
end


NSApplication.sharedApplication
NSApp.setDelegate(App.alloc.init)
NSApp.run
```

Line ❶ fetches a reference to the global status bar. Then line ❷ allocates screen space (on the left of all the other items) for the item you are about to create. The parameter NSVariableStatusItemLength says that the amount of space needed is unknown yet.

Line ❸ uses a handy class that represents an in-memory image. The next line, ❹, quits the program if there was no image file to load. Its **unless** check works because NSImage's init follows the Cocoa convention of returning nil—rather than the allocated object—when something goes wrong during initialization. There's a suitable image file in statusbar/stretch.tiff.

Finally, at ❺, the image is placed in the previously allocated space.

Try This Yourself

You can add text to the status bar with setTitle. Try that in statusbar-item.rb, both with and without an accompanying image.[3]

2.3 Menus

Our status bar item doesn't do anything, so let's give it a menu. For fun, I'll use it to make the app speak to us. That's not hard: I'll use a Cocoa object, NSSpeechSynthesizer, to turn text into speech.

Before starting that, let's separate concerns. App will concern itself only with application-wide events such as being launched and being terminated. A new class, SpeechController, will do everything else.

Here's the new version of App:

Download statusbar/speaking-statusbar.rb

```
class App < NSObject
  def applicationDidFinishLaunching(aNotification)
    statusbar = NSStatusBar.systemStatusBar
    status_item = statusbar.statusItemWithLength(NSVariableStatusItemLength)
    image = NSImage.alloc.initWithContentsOfFile("stretch.tiff")
    status_item.setImage(image)
❶   SpeechController.alloc.init.add_menu_to(status_item)
  end
end
```

Only one thing has changed, at line ❶. We just create a SpeechController, ask it to add its menu to the status bar item, and then forget about it. Notice that a SpeechController is an Objective-C object—you can tell because it's created with alloc and init.

And here's the SpeechController class:

Download statusbar/speaking-statusbar.rb

```
class SpeechController < NSObject
  def init
❶   super_init
    @synthesizer = NSSpeechSynthesizer.alloc.init
❷   self
  end
```

Like App, SpeechController descends from NSObject. A SpeechController needs to define its own init, though, because we want it to create an

3. If you're not working in the statusbar directory, get a copy of statusbar/stretch.tiff from there before running the script.

NSSpeechSynthesizer and hold onto it in an instance variable. Such an init method differs from Ruby's familiar initialize in two ways:

❶ In an ordinary Ruby class, the initialize method uses super to call its superclass's initialize method. In an NSObject subclass, init calls the superclass's init method with super_init. (In general, any overriding method *method* calls its superclass version with super_*method*.)

As you saw on page 26, init methods can sometimes return nil. For that reason, a pedantically safe use of the superclass would look like this:

```
return nil unless super_init
```

In this case, though, I know that NSObject's init always returns self. (In fact, it does nothing *but* return self, so I could omit the line entirely.)

❷ In an ordinary Ruby class, initialize's return value is irrelevant. In contrast, an NSObject subclass *must* return self (or, in the case of error, nil). If I'd forgotten line ❷, code like this:

```
s = SpeechController.alloc.init
s.add_menu_to(status_item)
```

. . . would make s an NSSpeechSynthesizer and then blow up on the next line with a "no such message" failure. Even after seeing a lot of those failures, it still sometimes takes me much too long to think of blaming init.

Now for the menu. In Cocoa, a menu is represented by an NSMenu that contains NSMenuItem objects. It's those objects that receive "you've been clicked" events from the window manager. If an NSMenuItem handles the event, it forwards the work by calling an *action method* attached to a *target object*. (See Figure 2.2, on the following page.)

The NSMenu itself does only a little work. It asks each item for its name and *key equivalent* (the keystroke that selects that item via the keyboard instead of the mouse). Then it paints all the items on the screen.

SpeechController's add_menu_to, shown in Figure 2.3, on the next page, wires all this together.

It begins (❶) by allocating an NSMenu object and attaching it to whatever container was given. This is another example of duck typing (and a benefit of separation of concerns): this particular class doesn't care

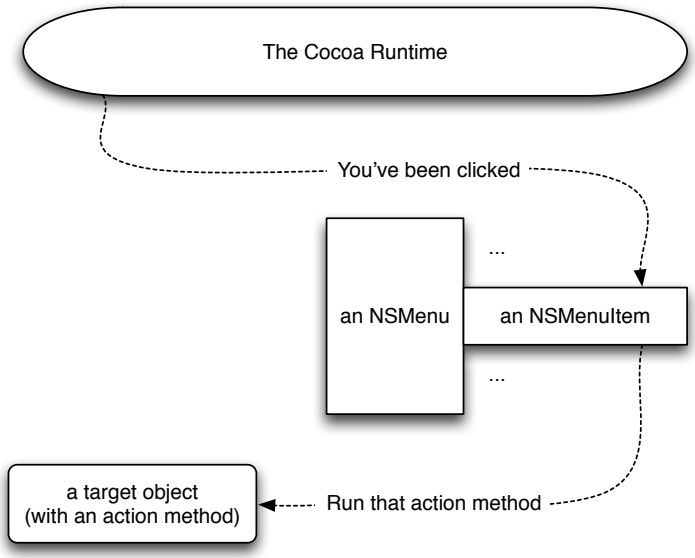


Figure 2.2: Clicking a menu

Download statusbar/speaking-statusbar.rb

```
  def add_menu_to(container)
❶   menu = NSMenu.alloc.init
    container.setMenu(menu)

❷   menu_item = menu.addItemWithTitle_action_keyEquivalent(
                        "Speak", "speak:", '')
❸   menu_item.setTarget(self)

    menu_item = menu.addItemWithTitle_action_keyEquivalent(
❹                       "Quit", "terminate:", 'q')
❺   menu_item.setKeyEquivalentModifierMask(NSCommandKeyMask)
❻   menu_item.setTarget(NSApp)
  end

❼ def speak(sender)
    @synthesizer.startSpeakingString("I have nothing to say.")
  end
end
```

statusbar/speaking-statusbar.rb

Figure 2.3: Building a menu

what it's attached to, so long as that object responds to setMenu. Today, it's a status bar item. Tomorrow, it could be something else.

Next, an NSMenuItem is created and assigned to the menu by addItemWithTitle_action_keyEquivalent (line ❷). What's up with *that* name? Objective-C has an interesting and nearly unique way of naming methods. Here's (almost) what Objective-C code that added a menu item would look like:[4]

```
[menu addItemWithTitle: "Speak" action: "speak:" keyEquivalent: ""]
```

The method being called here is named addItemWithTitle:action:keyEquivalent:. It takes exactly three arguments that have to come in exactly the defined order.

RubyCocoa has to provide you with a way of naming that Objective-C method. It can't use the same name, because method names in Ruby can't contain colons. So, the colons are replaced with underscores. To avoid excessive ugliness, you can leave off the last underscore, as I did at line ❷.[5]

The first and third arguments to the method provide the name and key equivalent. (This particular item has no key equivalent.) The second argument is the name of the message to send when the menu item is selected. Although the speak method is defined in Ruby, I've used Objective-C's notion of its name: *"speak:"*. The name ends in a colon because (as you'll see shortly), speak takes a single argument.

Which object receives the speak: message is set on the next line (❸). In this case, the SpeechController handles the message itself.

Lines ❹ and ❺ show how you create a keyboard equivalent. Those are almost never plain characters like q. They're usually characters with modifiers, like Command-Q. For whatever reason, the character and modifier keystrokes are set in separate methods.

The menu item will send a terminate: message, but not to SpeechController. Since it's a message about the whole app, it's targeted at NSApp (line ❻), an Objective-C class that implements terminate:.

4. I've removed a little type casting because it's not important to this explanation. To be pedantic, the title and key equivalent shouldn't be strings. They should be NSString objects, which are written as *@"string"*. Similarly, the action argument should be a "selector," not a string. You'll see more—and more correct—examples of Objective-C later in the book.

5. That's not always safe: consider an Objective-C class that has two methods, action and action:.

The speak (❼) action is simple. Notice that it takes a sender argument, which will be the NSMenuItem that was clicked. Action methods can use the sender to query or change the user interface.

If you run the app, you'll probably notice that the synthesizer takes a second or two to start talking after you click the menu item. Presumably it's doing some first-time initialization. It's more prompt the second time.

Try This Yourself

1. Put this at the end of speak:

   ```
   puts sender.objc_methods.grep(/title/i)
   ```

   Use one or more of those methods to change the menu after something is said.

2. While terminating, NSApp will send its delegate two messages: applicationShouldTerminate and applicationWillTerminate. The first lets the delegate decide to cancel shutdown, and the second gives it a chance to do any of its own cleanup.

   Use applicationWillTerminate to print out "Goodbye, cruel world!"

3. Make applicationShouldTerminate return false unless the app has spoken at least twice, true otherwise. See what happens when you return values like nil, *"fred"*, and the integer 0.

   A small quirk: unlike the delegate messages you've seen so far, applicationShouldTerminate takes an NSApplication as its argument, so sender or app would be a better name than aNotification.

   (If you need help, there's a solution in statusbar/speaking-statusbar-solution.rb.)

2.4 An Application Bundle

Fine though our script may be, it doesn't behave like a Mac application. If you double-click it, it doesn't launch. (Most likely, it opens in an editor.) It doesn't get an icon in the Dock, you can't see it if you Command-Tab through open applications, and so on. In this section, I'll explain what's special about apps. You'll create your first one in Chapter 3, *Working with Interface Builder and Xcode*, on page 37.

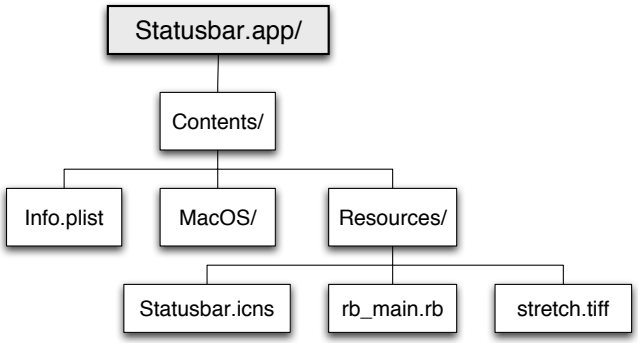


Figure 2.4: A bundle

On the Mac, executable code is delivered inside *bundles*. A bundle is just a directory hierarchy with a certain predefined structure. Applications are one kind of bundle. If you look inside statusbar/Statusbar.app, you'll see a structure like that of Figure 2.4.[6]

Reading roughly top to bottom and left to right, we have these files:

Contents/
: This identifies the bundle as a "modern" bundle.

Info.plist
: This contains various configuration options. For example, if you launch the application, you'll notice that it doesn't have a main menu and doesn't appear in the Dock. I made that happen by setting LSBackgroundOnly. Apple's *Runtime Configuration Guidelines* [App08x] has the gory details about all the options you can tweak.

MacOS/
: There is a small compiled Objective-C executable named Statusbar in this directory. It loads and starts the Ruby code. (You don't have to write the Objective-C yourself; Apple's Xcode, explained in Chapter 3, *Working with Interface Builder and Xcode*, on page 37, creates it for you.)

6. If you're browsing from the command line, Statusbar.app looks like what it is: a directory. If you're browsing with the Finder, it appears to be a single file. That's because it's a *package*, a special kind of directory that the Finder pretends is a file. All application bundles are packages. You can tell Finder to let you look inside it by selecting Show Package Contents from Statusbar.app's context menu.

Resources/
: This directory contains unchanging information the application might use. The good thing about resources is that their location is relative to the bundle itself, so code doesn't need to use absolute pathnames or know where it has been installed. Apple's *Resource Programming Guide* [App08v] teaches how to load and use resources.

  Because this is a small program, it has few resources:

  Statusbar.icns
  : .icns files contain the icons shown in the Finder, Dock, and so on. I create icon files with a drawing program and Icon Composer, which you can find in /Developer/Applications/Utilities.

  rb_main.rb
  : All your Ruby source files are stored as resources.

  stretch.tiff
  : Pictures, sound files, movies: all these are stored in Resources.

For more information about bundles, see Apple's *Bundle Programming Guide* [App08d].

Ruby Code Within a Bundle

Our existing script barely needs to change when moved into a bundle. Its name changed to rb_main.rb. (That's not required, but it's easiest to follow the convention.) The following old line of code won't work:

Download statusbar/speaking-statusbar.rb

```
image = NSImage.alloc.initWithContentsOfFile("stretch.tiff")
```

That line assumes the script is running in the same directory as the image file. That's not true for an application, which should fetch the image from Resources. That's done like this:

Download statusbar/Statusbar.app/Contents/Resources/rb_main.rb

```
image_name = NSBundle.mainBundle.pathForResource_ofType('stretch', 'tiff')
image = NSImage.alloc.initWithContentsOfFile(image_name)
```

Simple enough.

Try This Yourself

1. What's a running application's working directory? You can print it to the system log with this line:

   ```
   NSLog(Dir.pwd)
   ```

Add a menu item to Statusbar.app/Contents/Resources/rb_main.rb to do that. You can view the system log with the Console app (in /Application/Utilities). It will appear in the system.log log file.

Log the directory both when double-clicking the app and when opening it from the shell like this:

```
$ open Statusbar.app
```

2. What does the process environment look like when the app is launched by double-clicking (NSLog(ENV.inspect))?

   You should see that it's pretty sparse, containing nothing you set in your .bashrc file. Therefore, the common gem-loading trick of setting the environment variable RUBYOPT to *rubygems* won't work for a RubyCocoa app. Here are three solutions:

   a) Just use require 'rubygems' in your Ruby code. This is probably the best solution. In Leopard, RubyGems is guaranteed to be installed.

   b) Create a file named ~/.MacOSX/environment.plist. (Note that the directory starts with a period so that the Finder ignores it.) In it, place this XML:

   Download statusbar/environment.plist

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
             "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
   <plist version="1.0">
   <dict>
           <key>RUBYOPT</key>
           <string>rubygems</string>
   </dict>
   </plist>
   ```

   You will have to log out and log back in for the change to take effect. Once you do, the environment variable will be set for all applications.

   c) Add the following lines to Info.plist:

   ```
   <key>LSEnvironment</key>
   <dict>
     <key>RUBYOPT</key>
     <string>rubygems</string>
   </dict>
   ```

Beware, though: Info.plist is used only if your app is launched through Launch Services. The Finder and the shell's open command use Launch Services, but your IDE or programmer's editor may not.[7] In particular, Apple's Xcode does not.

2.5 What Now?

Given what you've seen, I hope you believe that writing a Mac app in Ruby is potentially as pleasant as any other Ruby programming. But only potentially. Three obstacles still have to be removed:

- Setting up even simple user interface elements like menus seems like fiddly, detail-heavy work. Imagine what windows with panels and sidebars and tooltips must be like! Apple's free developer tools remove that obstacle. They're explained in Chapter 3, *Working with Interface Builder and Xcode*, on page 37.
- The code has already used a number of mystery classes such as NSMenu, NSBundle, and NSImage. How do you find, understand, and use their documentation? That's explained in Section 3.2, *The API Browser*, on page 49.
- Information about classes and methods is all well and good, but classes and methods exist in a context. You need to know how Mac applications are put together and what kinds of things they can do. That's the topic of most of the rest of the book.

7. See Apple's *Launch Services Programming Guide* [App08o] for more.

Part I

A First Realistic App

Chapter 3

Working with Interface Builder and Xcode

In this chapter, we'll build a simple GUI that lets us fenestrate an app (open a window, or *fenestra*, into its internals). It's the acorn that will grow into the mighty oak tree of Section 1.7, *Our Example App*, on page 16. It looks like Figure 3.1. We'll imagine we'll type the name of the application we want to fenestrate in the smaller *text field*, and information from that app will ooze into the larger *text view*.

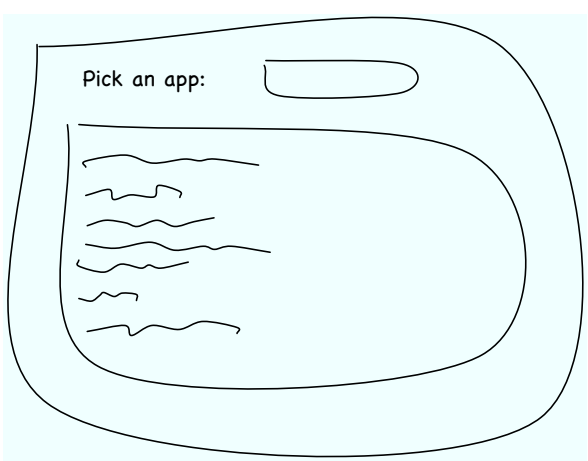
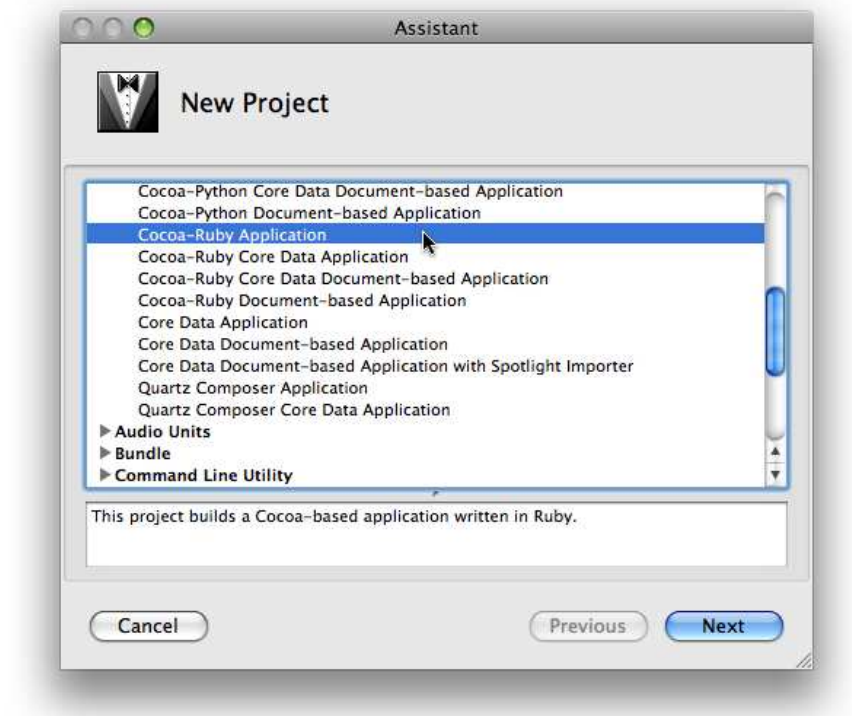


Figure 3.1: An app with input and output

3.1 The Basics

The two main tools for building Mac apps are *Xcode* and *Interface Builder*. (The latter is often abbreviated IB.) Xcode manages the collection of files used to build an app and provides a programmer's editor. It's akin to Eclipse or IntelliJ IDEA for Java, but it's tailored to Objective-C programs. Interface Builder is a tool for drawing executable user interfaces.

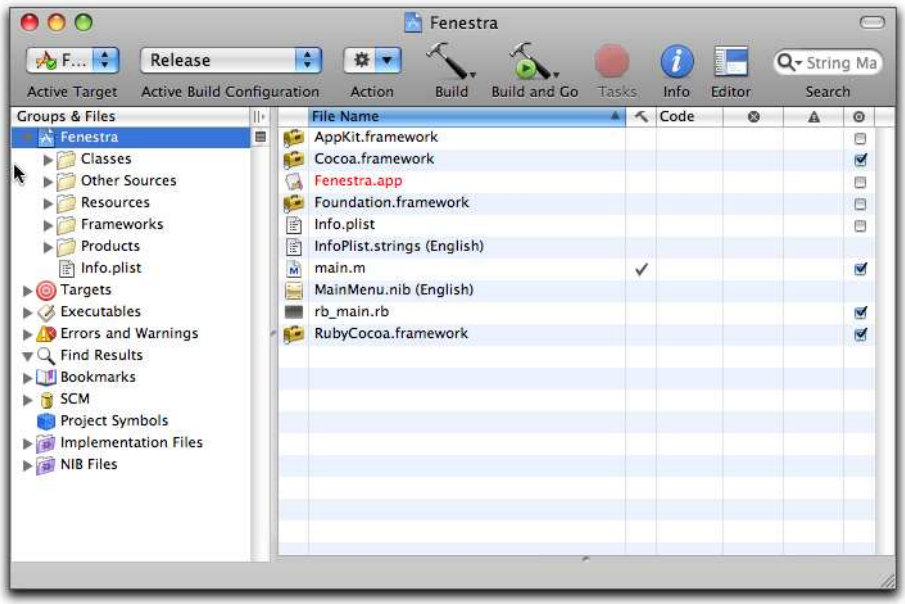
Begin by creating a project. Start Xcode (in /Developer/Applications). If you see a Welcome page, dismiss it (although you may want to look at it later). Create a project by selecting File > New Project. You'll see a screen like this:



Choose, as I did in the figure, a Cocoa-Ruby application, and hit Next.

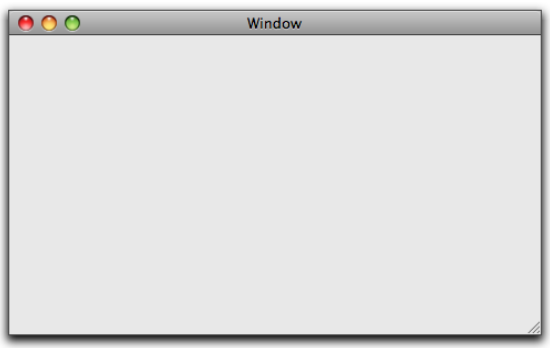
You'll be asked to name the project. Since you're also naming the app, it's conventional to pick a capitalized name. I picked "Fenestra."

After you hit Finish, you'll see something like the following:



This view has nothing to do with the file system structure of the project. It's more like a collection of smart folders in the Finder or iTunes. I won't give a detailed description of what you see, but notice some files we edited in Section 2.4, *An Application Bundle*, on page 31: Info.plist and rb_main.rb. Those are the source versions of the files that are put in the application bundle when it's built.

Click Build and Go on the toolbar now. After a bit of a pause, you should see this window:



Although you've written no code, your app already does some of what a real app does. You can hide and show it, minimize and zoom it, and quit it by either using the menu or pressing Command-Q. Quit it now.

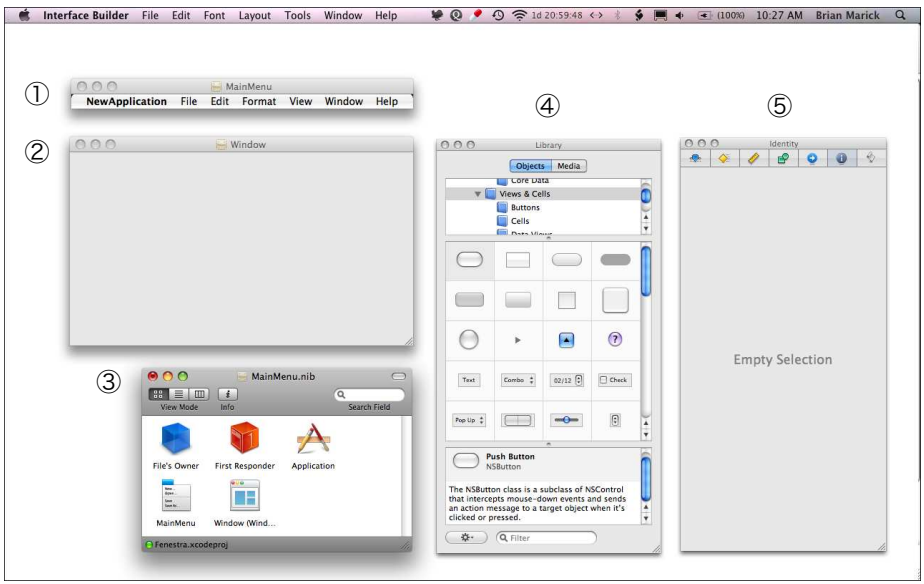
Our next job is to make that window have the label, text field, and text view shown in Figure 3.1, on page 37.

Interface Builder

In the project view, there's an entry in the right panel called MainMenu.nib. You can also see it under the Resources or NIB Files groups in the left panel. (If you see MainMenu.xib instead, don't worry; it's the same data, just formatted differently.)

You can think of a *nib file*[1] as describing a user interface, but the reality is more clever: it's actually a "frozen" ("marshaled" or "serialized" or "pickled") user interface. For more about that cleverness, see Section 11.2, *Archiving*, on page 134 and Section 14.4, *bind_toObject_withKeyPath_options*, on page 192. For now, edit the nib file by double-clicking its name.

You should see the following five windows, although sized and placed differently. (If you don't see all the windows, try selecting Window > Bring All to Front.)



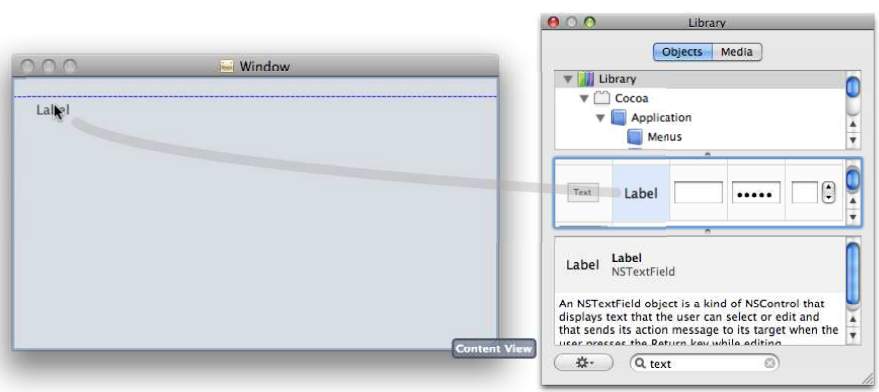
- The *main menu* is what appears in the menu bar on top of the screen. You'll be tinkering with that in Chapter 22, *Fit and Finish*, on page 287.

1. The name stands for "name's irrelevant, basically," although some claim it stands for "NeXT Interface Builder."

- The *main window* is what the window will look like in the running app.
- The *doc window* (or *document window*) represents the contents of the nib file in the same way that a window in Pages or Word represents the contents of a file. The doc window is a Finder-like view of the objects most important to the user interface. We won't be using all of them, and I'll describe only those we do. A good place to look for descriptions is in Apple's *Human Interface Guidelines* [App08b].

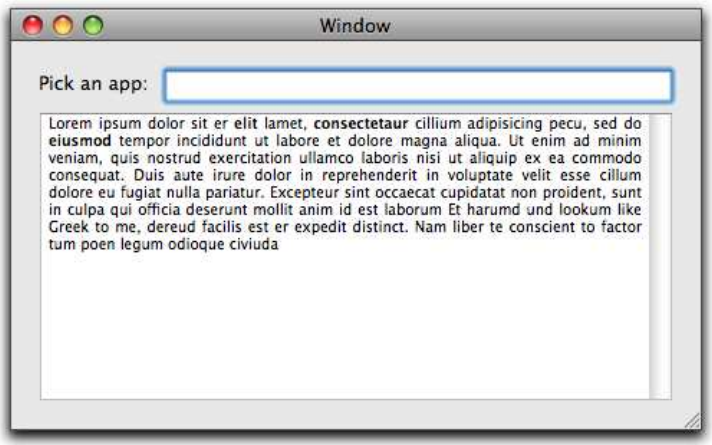
  The doc window starts out using icons, but hereafter I'll show it in list view because that lets me save some space in figures. I use list view in my ordinary programming, too—as the UI gets more complicated with objects within objects within objects, list view makes working with them easier.
- The *library* contains a large set of predefined user interface elements. Shortly, you'll be dragging three of them onto the main window. Dragging is how you build a Cocoa user interface with Interface Builder.
- The *inspector* is a tabbed editor for the bazillion-and-two things you can change about each UI element.

To begin, use the search box at the bottom of the library's window to search for *text*. One of the results will be an element named Label, as shown in the following images. Drag it into the top-left corner of the main window. Note that as you get close, guide lines will appear to help you put it an Apple-approved distance from the edges of the window.



After you drop the label, double-click it to edit its text. Change it to "Pick an app:" or whatever you like.

The library will also contain text fields and text views. Drag them into the window. Note that, when selected, they have drag handles that let you resize them. Make a window that looks something like the following:



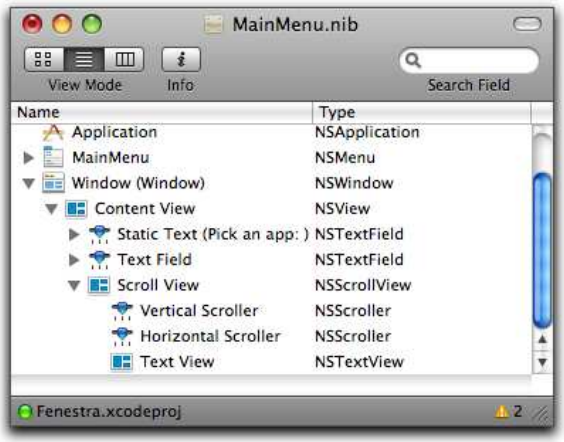
Big views have little views...

...in front of them to hide 'em. And little views have lesser views, and so ad infinitum.[2]

Everything you've done has involved two main classes of object. First, there's an NSWindow that represents a rectangular area on the screen. Within it are different subclasses of NSView, each responsible for its own piece of the window. Views are typically nested, with smaller views "on top of" larger ones (which means that they are responsible for the space they obscure).

2. With apologies to Augustus De Morgan and Jonathan Swift.

Once you've put IB's doc window in list mode, you can see the view structure of your window by expanding the NSWindow:



- The *content view* covers the entire space of the window. It contains all the other views.
- The label and text field cover part of the content view. Notice that they are both NSTextField objects—their different appearance is entirely because of how they're initialized. (See Section 3.5, *Attributes*, on page 58, for more.)
- What seems like a simple text view is actually its own hierarchy of objects. An NSScrollView contains the actual NSTextView and also two scrollers. One of them (the vertical scroller) takes up some space even when there's no need for a scrollbar, but the other is invisible until it's needed.
- If there were other visible objects in the window, even ones as insignificant as a vertical line used as a separator, they'd be NSView objects too. (A line is an NSBox, just a very, very thin one.)

You'll eventually need to know more about views to answer questions like "What object handles a mouse click?" and "What object handles keypresses?" This book will answer those questions only in the context of Fenestra. For complete details, see Apple's *Cocoa Event-Handling Guide* [App08h].

Connecting the Interface to Code

In the working application, typing into the text field will make a connection to a web app that will spew data for Fenestra to catch and display in various pleasing ways. For this first example, though, let's just have the text view echo whatever is typed into the text field.

If you squint, that looks a lot like the earlier status bar example (Section 2.3, *Menus*, on page 27). In it, some input (a mouse click in a menu) went to an *action*, a method inside the *target* class (SpeechController). That action did almost no processing; it just sent text to some Cocoa object attached to an instance variable. Like this:

In that code, @synthesizer is a particular kind of instance variable: one that points to some Cocoa object used for output. In the jargon, that's an *outlet*.

The connections shown earlier were made with code we wrote, code like this:

```
menu_item = menu.addItemWithTitle_action_keyEquivalent(
                      "Speak", "speak:", '')
menu_item.setTarget(self)
```

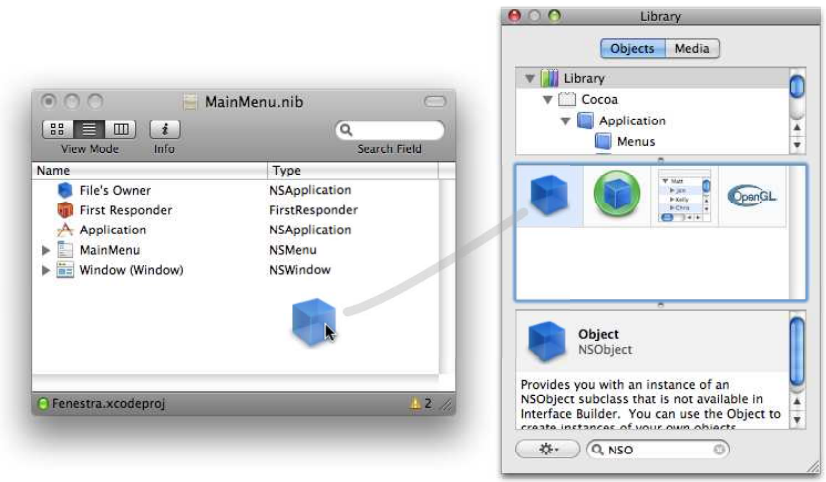
. . . and this:

```
@synthesizer = NSSpeechSynthesizer.alloc.init
```

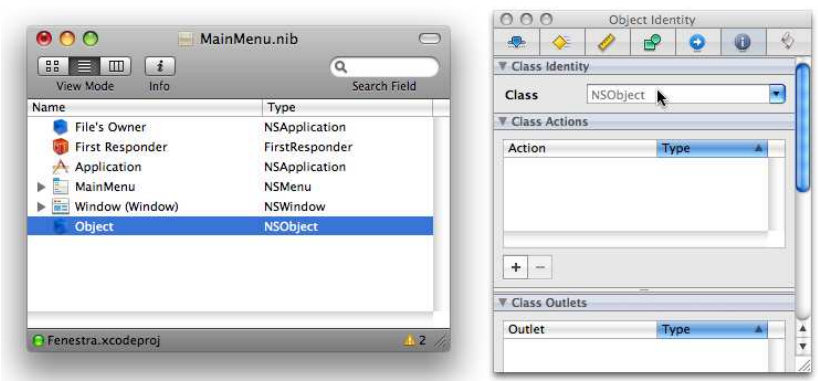
That kind of code is boring to write. Interface Builder can usually eliminate the need for it. All you need to do is create objects and draw lines between them. The objects and wiring instructions are stored in the nib file and decoded when the app starts up.

1. We already have almost all the objects that need to be wired together: you can see them in the main window and the doc window. Those are all UI objects, though, and we need one more to serve as the target of actions, the so-called *controller*.

The way you make an object in Interface Builder is by dragging some representation of it from the library. Since our class doesn't exist yet, it's not in the library. But no matter what the details, any object made in Interface Builder has to be an NSObject—all Cocoa objects are. So, we can use that as a placeholder. Search for NSObject in the library. When you find it, drag it into the doc window, as shown here:



2. Now we can add detail. With the new object selected, pick the Identity (sixth) tab in the inspector. The result should look like the following:



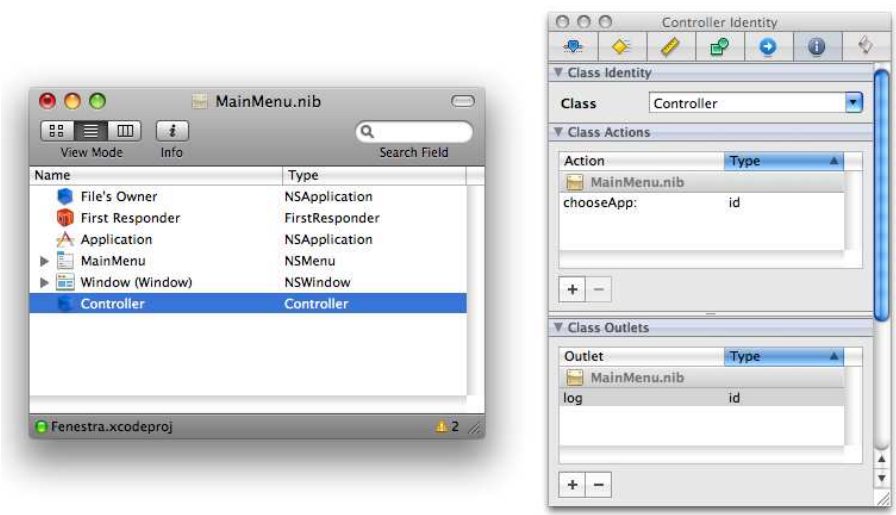
The Class Identity (topmost) field is where we tell IB about the class of our object. Controller class names conventionally end in Controller. Since we'll have only one for now, let's just call it Controller. Type that in the field. Notice that the name changes in the doc window after you hit Return.

3. In the Class Actions field, add a new action (using the + button). Name it chooseApp:. You have to end with the colon, since IB thinks the action is an Objective-C method that takes a single sender argument.

   The Type field is the type of the sender argument. A type of id means "figure it out at runtime," which both Ruby and Objective-C are perfectly happy to do. So, leave it as is.

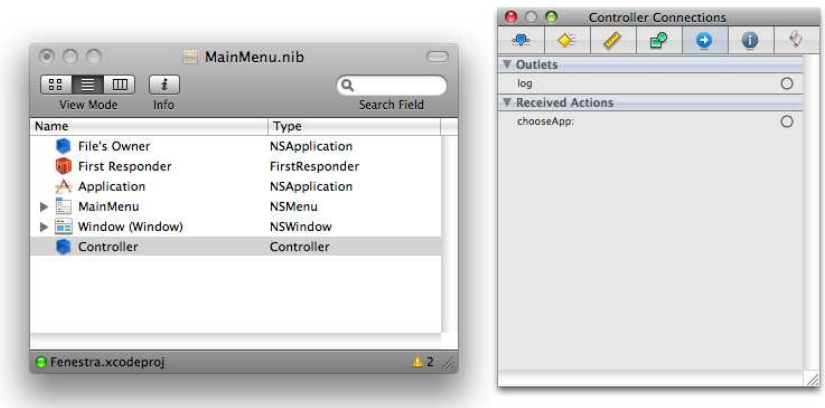
4. In the Class Outlets field, add an outlet named log. As with the chooseApp: action, there's no need to change the type from id.

The inspector should now look like this:

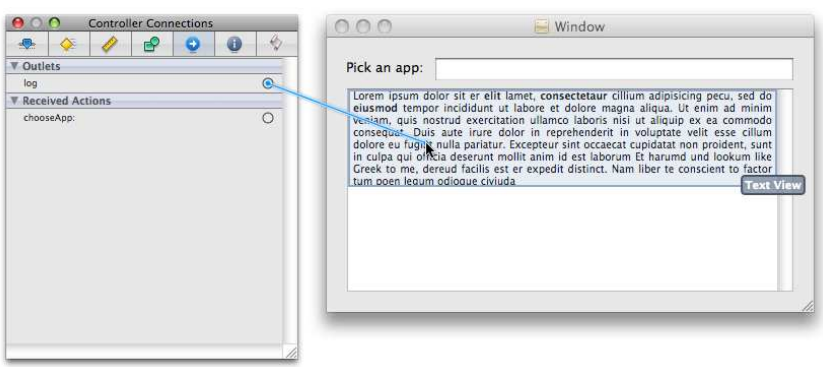


Interface Builder now knows that Controller exists (or will exist) and needs two connections.

1. First, open the Controller's Connections (fifth) tab in the inspector. That looks like this:



2. Our outlet and action show up there. Their lines have little circles at the end. If you click the one for the log outlet, you can drag a line to the text view in the main window, as shown in the following figures. Make sure to drop the end of the line in the "Lorem ipsum" text; if you drop it below there, you'll actually be making a connection to the scroll view that contains the text view. Check the tooltip to make sure.



In IB 3.0, the inspector loses track of what it's inspecting when you drop the far end of the connection. Worse, clicking the Controller in the doc window doesn't get it back. What I do is click some other object (like the text view) and then click the Controller again. (Later versions of IB don't have this problem.)

Once you've made the connection correctly, the log entry in the Connections tab will now describe the other end as "Text View," as shown here:

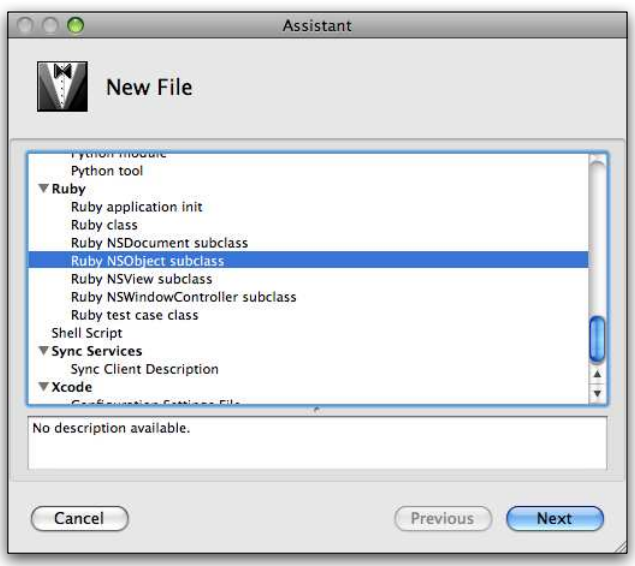
If you made a mistake, you can destroy the connection by clicking the little *x*. Double-clicking the now-filled-in circle will take you to the other end of the connection.

3. Next, drag the chooseApp: received action to its source in the text field.

IB now knows how Controller should be hooked into the UI. There's no code for that class yet, though. We'll write it in Xcode. Save the nib file with Command-S before switching from IB to Xcode.

3.2 Creating and Editing Classes in Xcode

In Xcode, select the File > New File menu item. You'll see this dialog box:



Select the Ruby NSObject subclass. Name the new file Controller.rb. Although that's unidiomatic Ruby, it seems to make Xcode happier.

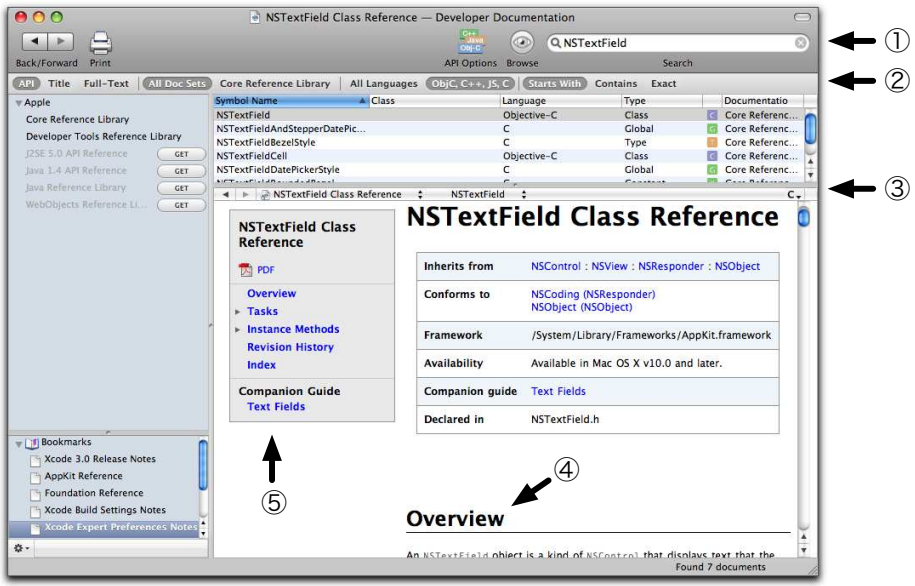
When that's finished, you'll see something like this:

(The comment at the head of the file will look different. If you want to peek ahead and see how to make it include your name and company name, see Section 22.6, *The About Window*, on page 297.)

The API Browser

I'm about to walk you through the implementation, telling you which methods to call on which classes as we go. Because of that, you don't need to know how to use the Cocoa API browser to find classes and methods yet, but you may want to anyway.

The browser is available by selecting Help > Documentation. It looks like this:



The arrows point out the parts of the browser I find most helpful:

1. I use the search field to find documentation for classes and methods whose name I know. To go from "I need to do *X*" to "I need to use class *Y*," I use Google. I find it does a better job of finding relevant hits in Apple non-API documentation than does Xcode's search or the one at Apple's Developer Connection website.[3]

2. If you have a similar experience (which you might not), you can restrict search results to the API with the button on the far left of this bar. Toward the middle, there's an All Languages button, which means in practice "include the Java API." That's not useful, so I turn it off.

3. In the middle of this bar, there's a drop-down list that contains the class name. If you open it, you'll get quick access to instance and class methods.

   At the right, marked with a "C," you can get quick access to superclass and subclass documentation.

3. http://developer.apple.com/

But I Already *Have* an Editor!

If you're a typical reader, you already have a favorite editor that you use for Ruby, and you're not wild about changing to Xcode. Chapter 10, *Project Organization, Builds, and Your Favorite Editor*, on page 117, will tell you how to do your editing and building without Xcode. You'll likely want to keep using Xcode for infrequent actions such as adding new nib files to a project. To build up enough familiarity with Xcode, I recommend that you continue using it for everything—just for now.

4. Each class begins with an overview. I've found them quite good, and I make a habit of reading them.
5. There are often companion guides that put the class in context. I usually at least skim them.

Implementation

Implement your new class as shown in Figure 3.2, on the following page:

❶ In Interface Builder, we declared that Controller had a log outlet. From inside the class, the outlet is just the instance variable @log.

It's the responsibility of code outside this class, code we don't write, to initialize @log. For that reason, we have to declare an attr_writer method for that code to call.

❷ The text view's default content will be the strange "Lorem ipsum" text that Interface Builder showed you. I want to replace that with an empty string. That means sending a message via @log—but only after the attr_writer method has been used to initialize it. The Cocoa runtime calls awakeFromNib after all outlets are guaranteed to be connected, so that's the method that should initialize the text view. However, as is so often the case in object-oriented programs, awakeFromNib doesn't do anything itself; it just passes work along to someone else.

❸ chooseApp implements the action we declared in Interface Builder. Unlike @log, we don't need to provide any special access to the outside world—it's just a public method like any other.

Download fenestra/text-field-to-view/Controller.rb

```
require 'osx/cocoa'

class Controller < OSX::NSObject
  include OSX

❶   attr_writer :log

❷   def awakeFromNib
      record('')
    end

❸   def chooseApp(sender)
      record(sender.stringValue)
    end

    def record(string)
❹     everything = NSRange.new(0, @log.textStorage.length)
❺     @log.replaceCharactersInRange_withString(everything, string)
    end
end
```

fenestra/text-field-to-view/Controller.rb

Figure 3.2: The first controller

chooseApp shows how you retrieve values from simple views like text fields: you just ask for their stringValue. As with awakeFromNib, the real work will be done by record.

❹ A text view is part of Cocoa's text subsystem, which is both powerful and complex. Among its complexities are various ways to get at the text inside a view. This code shows one way.

This way of working with a text view uses an NSRange. You can think of NSRange classes like selections you make with a mouse; indeed, if you ask a text view for its selection, you'll get back an NSRange. Like a selection, an NSRange can describe a run of one or more characters. In that case, inserting new text will replace the old text. Alternately, an NSRange can describe zero characters. (Think of a cursor blinking between two characters.) In that case, inserting new text doesn't change old text.

In this case, we want to replace all the text in the text view. We can do that by creating an NSRange that starts at character zero and extends for the length of the text. You can't ask an NSTextView

object for its text length; instead, you have to reach into the view, pull out the associated textStorage, and ask that object.

If you're wondering why an NSRange is created with new instead of alloc. . . hold that thought for a moment.

❺ replaceCharactersInRange_withString does what its name says.

The absence of explicit type declarations in Ruby hides something about record. When it's called from awakeFromNib, it's given a plain Ruby string (class String). When it's called from chooseApp, it's given the stringValue of a text field, which is a Cocoa NSString. So, replaceCharactersInRange_withString is sometimes given a String and sometimes an NSString. Is that safe?

If the Objective-C method replaceCharactersInRange_withString actually *received* a Ruby String, the result would be ugly—very ugly. RubyCocoa saves us from disaster by converting any String passed to an Objective-C method into an NSString before it reaches the method. You'll see more about conversions throughout the book. If you're impatient, I've provided a command, conversions/round-trip, that lets you convert Ruby objects and examine their Objective-C equivalents. Here's an example:

```
$ cd conversions
$ ./round-trip 1
------
If 'In' is the Fixnum '1' (1),
then 'Out' will be the OSX::NSCFNumber '1' (#<NSCFNumber 1>) with
classes [OSX::NSCFNumber, OSX::NSNumber, OSX::NSValue, OSX::NSObject]
Will Out==In? Yes.
Out.to_ruby will be the Fixnum '1' (1).
```

Use round-trip -h to find out more.

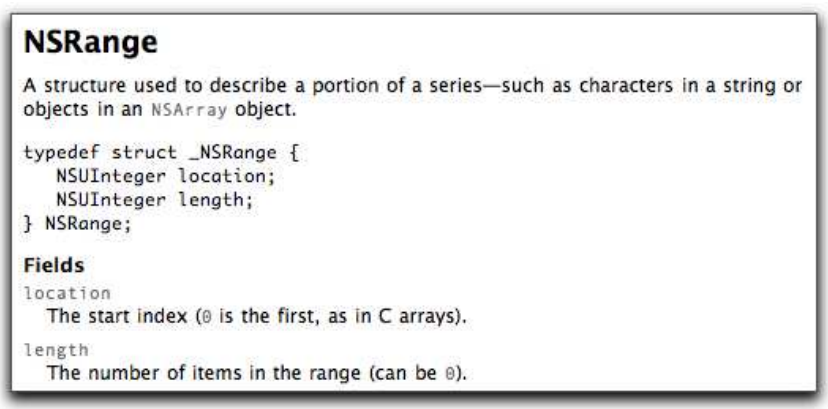
Return values *from* Cocoa methods are not converted. (If they were, the Ruby code couldn't modify the original Cocoa object.) However, in the case of classes that have close Ruby equivalents, such as NSString, most of the corresponding Ruby class's methods will work. For example, you can add two NSString classes together or add an NSString to a Ruby string. If a method doesn't work, you can convert the object into its Ruby equivalent, either with normal Ruby methods like to_s or to_hash or with the special method to_ruby. To convert a Ruby object to its Cocoa equivalent, use to_ns.

❹ (again)

The odd thing about NSRange is that its name suggests that it's a descendent of NSObject, but the use of new to create one (rather than alloc.init) suggests that it's not—that it's a pure Ruby class. Which is it?

It's a pure Ruby class. Like a String, an NSRange is converted on its way into an Objective-C method. The difference is that an NSRange isn't converted into an Objective-C *object*; it's converted into an Objective-C *struct*. If you're familiar with the C language or its derivatives, you know what a struct is. If you're not, just think of it as an object that contains no interesting methods, only data.

I expect there'll be only one time you'll ever care about the difference between a struct and a class. You'll be coding away when you discover you need to use replaceCharactersInRange_withString. You'll look at its documentation, find that it takes an NSRange argument, wonder how you create one of those, click the link that goes to NSRange's documentation, and discover this:



NSRange

A structure used to describe a portion of a series—such as characters in a string or objects in an `NSArray` object.

```
typedef struct _NSRange {
   NSUInteger location;
   NSUInteger length;
} NSRange;
```

Fields

`location`
The start index (`0` is the first, as in C arrays).

`length`
The number of items in the range (can be `0`).

That looks different from a class's documentation, but it tells you the facts you need to know:

- You create one of them with new, giving *location* and *length* arguments (in that order).
- If you need to extract the location or length from an NSRange, you'll use the method location or length.

Running the App

Before running the app for the first time, open the debugger console window by selecting Run > Console. That's where Ruby's error output goes, and you'll appreciate having it when you make a typo in your code. Because the console window has an annoying way of shuffling itself behind other Xcode windows, its hotkey was the first I memorized: Command-Shift-R. You can also ask Xcode to pop up the console whenever it launches the app. Set On Start to Show Console in the Debugging preference panel (Xcode > Preferences).

Run the app from the toolbar (Build and Go). Be careful always to use Build and Go when you change a Ruby file. If you don't, the changed file won't be copied into the app bundle build/Release/Fenestra.app, which is what Go runs.[4] Type something into the text field, hit Return, and observe with wonder that the characters are echoed into the text view. If that's not what happens, see Section 3.3, *Debugging*.

3.3 Debugging

There's a good chance your first app has already failed. If not, you're sure to make a mistake soon, so now is a good time to introduce debugging a RubyCocoa app. For simplicity's sake, add a typo to the app in fenestra/text-field-to-view. You can edit that project in Xcode by double-clicking Fenestra.xcodeproj in the Finder (or typing open Fenestra.xcodeproj to the shell).

Here's the typo: change stringValue to stingValue. Build and run again, type something in Fenestra's text field, hit Return, and then look at the console log. I expect you to see the following, except that I've truncated the immensely long pathnames:

```
2008-04-07 16:02:26.985 Fenestra[24870:10b] Controller#chooseApp: OSX↩
::OCMessageSendException: Can't get Objective-C method signature for ↩
selector 'stingValue' of receiver #<OSX::NSTextField:0x242a52 class='↩
NSTextField' id=0x3e05b0>
        .../oc_wrapper.rb:50:in `ocm_send'
        ...
        from .../Controller.rb:15:in `chooseApp'
        from .../rb_main.rb:22:in `NSApplicationMain'
        from .../rb_main.rb:22
```

4. Another gotcha is that if you rename a Ruby file, the old version won't be deleted from the build directory. Since rb_main.rb loads all the Ruby files, you'll get two versions of the code. Use Build > Clean before building.

All but the first line is the Ruby call stack that you've seen time after time, typo after typo, mistake after mistake.[5] The first line is the same Objective-C "no such method" error you saw on page 25.

Now let's make a more subtle error. Simple elements like buttons and text fields can provide more than strings. They can also return integers and floating-point numbers. Change stingValue to intValue. Build and run, and then type some valid number into the text field.

I expect you to see the following:

```
2008-04-08 15:24:41.159 Fenestra[27739:10b] *** -[NSCFNumber length]: ↩
 unrecognized selector sent to instance 0x2324a0
2008-04-08 15:24:41.161 Fenestra[27739:10b] Controller#chooseApp: OSX ↩
❶ ::OCException: NSInvalidArgumentException - ***-[NSCFNumber length]:  ↩
unrecognized selector sent to instance 0x2324a0
❷         .../oc_wrapper.rb:50:in `ocm_send'
          .../oc_wrapper.rb:50:in `method_missing'
          .../oc_attachments.rb:61:in `objc_method_missing'
          .../oc_attachments.rb:61:in `method_missing'
❸         .../Controller.rb:7:in `record'
          .../Controller.rb:15:in `chooseApp'
          .../rb_main.rb:22:in `NSApplicationMain'
          .../rb_main.rb:22
```

Here's how you might use the stack trace. We're inside the Ruby method record (❸). We drop into the Objective-C universe at ❷. We can't see what work was done after that—we could have been twenty nested methods deep when some sender sent some receiver the length message. The error message (❶) does tell us that the intended receiver of the message is an NSCFNumber.[6] So. . . we're asking a number of some sort for its length, and apparently numbers don't have lengths. record claims it takes a string argument, but it must have been passed a number from chooseApp. And indeed it was: bug found.

Return to the stringValue version for the rest of this chapter.

The Time-Honored Tradition

Ruby's standard output also goes to the console. If you're used to debugging by sprinkling puts calls throughout your code, you can still do that. NSLog is an alternative to puts. It will go to both the Xcode console and the system console. It also includes a timestamp and the name

5. If you've never seen one before, you're inhumanly good. Put this book down. You have a world to run.
6. What Ruby calls *messages* or *method names*, Objective-C calls *selectors*.

of the source app. For Xcode debugging, I find that just clutters up the log, so I usually use puts.

Try This Yourself

1. Change the spelling of chooseApp in the Ruby code, but leave it as is in the nib file. Build and run. What happens?
2. Change the spelling of log in the Ruby code, but leave it as is in the nib file. Build and run. What happens?
3. Someday, you'll make a mistake and use alloc.init on a Ruby class that corresponds to a struct. Try that now with NSRange. What happens?

3.4 Synchronizing Interface Builder and Xcode

It's a little awkward to have to type each outlet and action name to both Interface Builder and Xcode. For your greater comfort, IB can find names in source code if you declare them specially. Change the beginning of Controller to add the special declarations and to add a new outlet:

Download fenestra/text-field-to-view-and-title/Controller.rb

```
class Controller < OSX::NSObject
  include OSX

  ib_outlet :log, :logWindow
  ib_action :chooseApp
```

These declarations don't change the app's behavior at all. ib_action does nothing, and ib_outlet is the same as attr_writer. It's true that we've added a setter for a new instance variable, @logWindow, but we don't use it yet.

To see where the declarations *do* have an effect, open Interface Builder. (Double-click MainMenu.nib.) Then select Controller in the Finder-like doc window. In the inspector, navigate to the Connections (fifth) tab.

You should now see this:

If you don't, make sure you've saved your Xcode changes and use IB's File > Synchronize With Xcode menu item.[7]

Now connect the logWindow outlet to the entire window. You can either drag from the logWindow's circle in the inspector to the title bar of the main window or drag to the Window entry in the Finder-like doc window.

To see the new outlet in action, change chooseApp to echo the target app's name into the window's title bar:

Download fenestra/text-field-to-view-and-title/Controller.rb

```
def chooseApp(sender)
  entered = sender.stringValue
  record(entered)
❶ @logWindow.title = entered
end
```

Notice the assignment operator at ❶. I could have set the window's title with an explicit method call like this:

```
@logWindow.setTitle(entered)
```

. . . but RubyCocoa is kind enough to convert assignment into that for me.

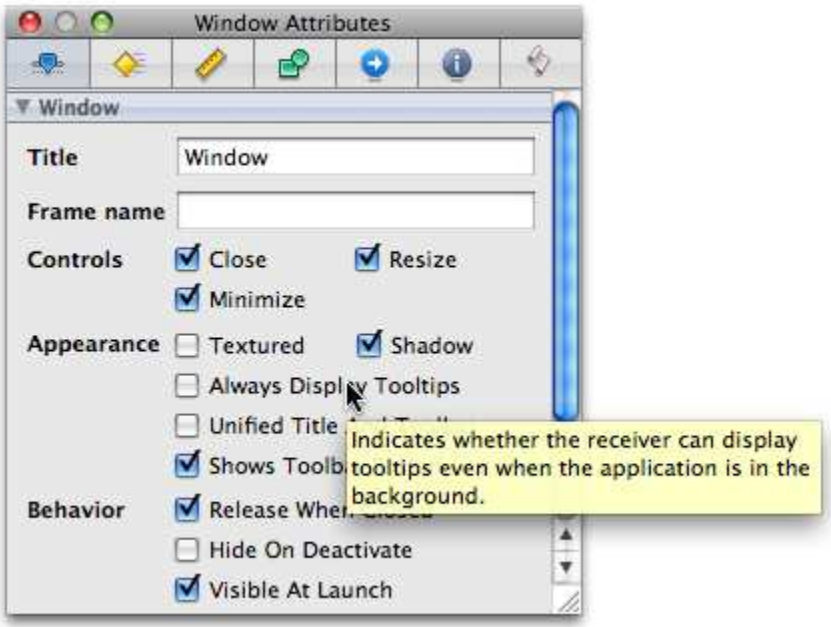
Give the new app a try. With luck, typing in the text field will cause text to appear in two places.[8] There's still something ugly, though: when the app launches, the window's title is "Window." We'll fix that next.

3.5 Attributes

Right now, our app does nothing useful, but it will eventually open a fenestra to another app. So, a useful starting window title might be "- No App -." That title could be set in awakeFromNib, but it's more convenient to do it in Interface Builder. In IB, select the main window, and then go to the inspector's Attributes (first) tab.

7. This command has been removed from Xcode 3.1. IB synchronizes whenever you switch to it.
8. If you instead got "undefined method 'title=' for nil:NilClass," it's most likely you forgot to connect the logWindow outlet in Interface Builder.

You should see something much like this:



Change the Title attribute to say "- No App -" (or whatever title you prefer). Save; then build and run.

Cocoa UI objects have an incredible variety of attributes. Some of them, like Title, you'll find easy to understand. Others you'll learn as you work through this book. For the remainder, try looking at tooltips. If that doesn't work, dive into the sources of documentation listed in Section 1.11, *Solving Problems*, on page 19.

Try This Yourself

1. Change the text view to make it uneditable. You might change the background color, too, as a signal that it can't be edited.
2. Change the label's attributes to make it look like a text field. What happens if you type in it and then press Return?

3.6 Overriding Window Behavior with a Delegate

Here's what the *Apple Human Interface Guidelines* [App08b] say about applications like ours:

> In most cases, applications that are not document-based should quit when the main window is closed. For example, System Preferences quits if the user closes the window. If an application continues to perform some function when the main window is closed, however, it may be appropriate to leave it running when the main window is closed. For example, iTunes continues to play when the user closes the main window.

Right now, Fenestra manages no documents, has a single window, does no processing after that window closes, and has no way to reopen the window if the user does close it. So, let's make it exit by having the window tell the controller (via delegation) when it closes. The controller can then send the terminate message to NSApp.

You could set the window's delegate in awakeFromNib, like this:

Download fenestra/autoclose/Controller.rb

```
def awakeFromNib
  @logWindow.delegate = self
  record('')
end
```

However, it's more idiomatic to set up unchanging relationships in IB. IB considers an object's delegate to be one of its outlets, and you set the delegate in the Connections inspector:

In either case, the Controller should volunteer to handle a "window is about to close" event by defining this method:

Download fenestra/autoclose/Controller.rb

```
def windowWillClose(notification)
  NSApp.terminate(self)
end
```

3.7 Try This Yourself

1. Instead of making the app quit when its window closes, prevent the window from closing. (Hint: look at the window's attributes.) What now happens when you try Command-W?
2. Change the code to append new text onto the end of the text view instead of replacing its contents. The NSRange corresponding to no selection with the cursor at the end of the text is one whose length is zero and whose starting position is @log.textStorage.length.

3.8 What Now?

It's now time to turn the application into something minimally useful by having it receive and display interprocess NSNotifications from a web app. After that, we'll have an app that works but has some glaring flaws.

Chapter 4

One Good App Observes Another

Now that Fenestra has a (barely) tolerable interface, it's time to work on the code behind it. We'll use Cocoa's notification system for communication between Fenestra and some other app. Because notifications are widely used in Cocoa apps, I'll describe them in more detail than is needed for just this one app.

4.1 Notifications Within an App

The version of our app in Section 3.6, *Overriding Window Behavior with a Delegate*, on page 60, works because some object[1] follows the main window's delegate link to a Controller, notices that it defines windowWillClose, and calls that method, giving it a chance to make the app exit.

Controller can learn about the window closing in another way. It can subscribe to notifications from the NSWindow:

Download fenestra/autoclose-with-notifications/Controller.rb

```
def awakeFromNib
  center = NSNotificationCenter.defaultCenter
  center.addObserver_selector_name_object(self, :windowWillClose,
                                          'NSWindowWillCloseNotification',
                                          @logWindow)
  record('')
end
```

In words, our Controller is saying, "Hey! Default notification center! At some point, the object I know as @logWindow might announce that it's going to close. If so, send the windowWillClose message to a particular

1. It's not actually the main window's NSWindow itself, but it might as well be.

object (namely, me).” The windowWillClose method is unchanged from the one in the previous version (found on page 61):

Download fenestra/autoclose-with-notifications/Controller.rb

```
def windowWillClose(notification)
  NSApp.terminate(self)
end
```

I could have named the method something else (which you can’t with a delegate), but when it comes to windows, closing delegation and observing are just different ways of getting the same information to our Controller, so using the same name seems appropriate. (Even windowWillClose’s argument is the same. It’s an NSNotification object, described in Section 4.1, *The Finer Points of Notifications*, on the following page.)

You can see the new version working by first using IB to turn off the window’s delegate outlet (click the little *x* if the outlet shows a connection) and then building and running.

Delegation vs. Notification

You have now seen two different code designs for window closing. You might ask, which is better? Since they both do the same thing, I embrace my inner slacker and ask two questions:

- Which is less work today?
- Which will be less work in the future?

I personally place more weight on the first question because I get its answer right more often.

Setting a delegate requires drawing a line in Interface Builder. Adding a notification means typing code in awakeFromNib. For me, delegation wins.

Looking to the future, I can imagine myself adding another window to the app. After that, Fenestra should behave like other multiwindow Mac apps: closing a window just closes the window. If you want to exit, do that explicitly.

I know my imaginary future self all too well: he doubtless will have forgotten both how and where I implemented the current behavior. If I used delegation, all my future self will have to do is look for the window’s delegate in IB, hop to that class, and find method windowWillClose. If I used notifications, finding the code would be more work. First, I’d have the wasted work of checking for the delegate; on top of that, I’d have to grovel around in the code to find a method that wouldn’t necessarily even be called windowWillClose.

Try This Yourself

Have windowWillClose print *"I'm here!"* to the console. Either puts or NSLog work fine. Then:

1. Build and run the app. Close the window (Command-W).
2. Run the app again. Quit the program (Command-Q).
3. Reestablish the delegate link between the main window and the Controller, but continue to make the Controller an observer of the @logWindow. Build and run. Exit by closing a window.

What do you think is happening?

I think that NSApplication's terminate method closes all open windows. In our case, terminate is called because an open window is closing. So, terminate blithely closes that window again. windowWillClose again calls terminate. Fortunately, terminate is smart enough not to go any further down the rat hole.

It also seems that delegation to windowWillClose is independent of notification delivery. So, in the third case, terminate is called three times: once because of delegation, once because of notification, and once because terminate closes all windows.

The Finer Points of Notifications

"Don't care" values

In the previous example, the code asked to hear about notifications named NSWindowWillCloseNotification. A nil argument asks to hear about notifications with any name. Try that, printing the notifications with puts notification; then try various window operations (such as minimizing and hiding) to see what notifications get sent. To see a complete list, use the NSWindow class reference.

If you use nil for the object to observe, you'll observe all objects that send a particular named notification. In our case, that's not interesting, since only windows send NSWindowWillCloseNotifications, and we have only one window. You can, however, give nil as both the name and object arguments. Then you see *all* notifications from *all* objects. Try that to see how many notifications are sent in even the simplest Cocoa applications. (If you use windowWillClose to print out the notifications, comment out the line that calls terminate.)

userInfo arguments

If you tried the change in the previous paragraph, you probably saw output with extra information. Here is the notification, for

example, that comes from hitting Tab or Return to finish editing in a text field:

```
NSConcreteNotification 0x3c9330 {name = NSControlTextDidEndEditin⟷
gNotification; object = <NSTextField: 0x3e15c0>; userInfo = {
    NSFieldEditor = <NSTextView: 0x2307490>
    Frame = {{2.00, 3.00}, {271.00, 17.00}}, Bounds = {{0.00, 0.0⟷
0}, {271.00, 17.00}}
    Horizontally resizable: YES, Vertically resizable: YES
    MinSize = {271.00, 17.00}, MaxSize = {40000.00, 40000.00}
;
    NSTextMovement = 16;
}}
```

Each notification can pass along an NSDictionary. NSDictionary is Cocoa's equivalent of Ruby's Hash: a collection of key/value pairs. When printed, an NSDictionary looks something like a hash, but not exactly. Keys and values are separated by =,, not =>, and strings aren't enclosed in quotes.

If you were writing code in Objective-C, you'd retrieve NSDictionary values like this:

```
[dictionary objectForKey: key]
```

You can do the same in Ruby if you want:

```
dictionary.objectForKey(key)
```

But ordinary hash notation also works:

```
dictionary[key]
```

Beware, though: not all Hash methods will work on an NSDictionary.

The name and sender

A notification contains its name and a pointer to the object that sent it. They're retrieved like this:

```
notification.name
notification.object
```

Sending notifications

You send a notification like this:

Download notifications/examples/within-process-userinfo.rb

```
Center.postNotificationName_object_userInfo("notification name",
                                             self,
                                             {'string' => 'world',
                                               'int' => 5,
                                               'array' => ARGV})
```

(To save horizontal space, I've defined constant Center to be the same NSNotificationCenter.defaultCenter we've already seen.)

If you have no userInfo to add, use a slightly different method:

Download notifications/examples/within-process.rb

```
Center.postNotificationName_object("notification name", self)
```

In cases where receivers aren't expected to care which object sent the notification, programmers sometimes use the *object* argument to send data that more properly should go into *userInfo*. That is, rather than writing this:

```
Center.postNotificationName_object_userInfo("got argv",
                                             self,
                                             { "argv" => ARGV })
```

. . . they'll write the following:

```
Center.postNotificationName_object("got argv", ARGV)
```

Conversions

When you create a notification, you'll likely use Ruby objects such as strings, integers, arrays, and nested hashes to fill its *userInfo*. When it's received, though, the Ruby objects have all been converted to their Objective-C equivalents: NSStrings, NSNumbers, NSArrays, and nested NSDictionary objects:

```
% ruby within-process-userinfo.rb with args
=== Looks innocent enough when you 'to_s' it:
NSConcreteNotification 0x57ae90 {name = notification name; objec⟵
t = <Sender: 0x2a9ce0>; userInfo = {
    array =     (
        with,
        args
    );
    int = 5;
    string = world;
}}

=== ... but those are not simple Ruby objects:
#<NSCFDictionary {#<NSCFString "int">=>#<NSCFNumber 5>, #<NSCFSt⟵
ring "array">=>#<NSCFArray [#<NSCFString "with">, #<NSCFString "⟵
args">]>, #<NSCFString "string">=>#<NSCFString "world">}>
```

For even more about notifications, see Apple's *Introduction to Notification Programming Topics* [App08q].

4.2 Notifications Between Apps

To show you notifications between apps, I've written two scripts for you. They're in the notifications/examples directory. To use them, open two terminal windows, and cd each to that directory. In one, type this:

```
$ ruby sender.rb
```

It sends one notification per second to whichever apps care. The notifications contain only a number, incremented each time. In the other shell, type this:

```
$ ruby receiver.rb
com.exampler.sender sez next number 7
com.exampler.sender sez next number 8
com.exampler.sender sez next number 9
```

receiver.rb prints the numbers. Notice that it doesn't start at 1 unless you're an amazingly fast typist. During the time it took for you to start the receiver, no app cared about the sender's notifications, so they were discarded.

Sending Distributed Notifications

Sending notifications between applications is pleasantly similar to sending them within applications. Our sender is shown in Figure 4.1, on the next page.

❶ Like ordinary notifications, distributed notifications are handled by a notification center. The only difference is the class: it's NSDistributedNotificationCenter instead of NSNotificationCenter.

❷ The method to send the notification has the same name. There are restrictions on the arguments, though.

1. The "object" that sends the message has to be a string that somehow names the sending app. To avoid conflicts, the reverse hostname convention is often used. (So, since I own exampler.com, I name apps "com.exampler.*appname*.")
2. The userInfo hash should use only simple objects such as numbers, strings, arrays, and nested hashes. For more details, see Apple's *Property List Programming Guide* [App08u] or Chapter 11, *Persistent User Preferences*, on page 127.

❸ Any object descended from NSObject can send distributed notifications. No special setup is required.

Download notifications/examples/sender.rb

```
#!/usr/bin/env ruby

require 'osx/cocoa'
include OSX

class Controller < NSObject
  def init
    super_init
    count = 100
    1.upto(count) do | i |
      puts "#{count - i} more notifications to post." if i % 25 == 0
      announce(i)
      sleep 1
    end
    puts "Sender will now stop posting notifications."
    exit
  end

  def announce(number)
❶   center = NSDistributedNotificationCenter.defaultCenter
    name = "next number"
    app = "com.exampler.sender"
❷   center.postNotificationName_object_userInfo(name, app,
                                                'value' => number)
  end
end

if $0 == __FILE__
❸   Controller.alloc.init
end
```

notifications/examples/sender.rb

Figure 4.1: Sending a notification

Download notifications/examples/receiver.rb

```
#!/usr/bin/env ruby

require 'osx/cocoa'
include OSX

class Controller < NSObject
  def init
    super_init
❶   center = NSDistributedNotificationCenter.defaultCenter
    center.addObserver_selector_name_object(self, :next_number,
                                            "next number",
                                            "com.exampler.sender")
    self
  end

  def next_number(notification)
    name = notification.name
    app = notification.object
    info = notification.userInfo
    number = info['value']
    puts "#{app} sez #{name} #{number}"
  end
end

if $0 == __FILE__
  Controller.alloc.init
  NSApplication.sharedApplication
❷   NSApp.run
end
```

notifications/examples/receiver.rb

Figure 4.2: Receiving a notification

Receiving Distributed Notifications

If sending a distributed notification is like sending an ordinary one, receiving one is even less different. See Figure 4.2.

❶ You have to use NSDistributedNotificationCenter.

❷ Unlike ordinary notifications, distributed notifications can be received only inside an app's run loop. If the app isn't running, no notifications will be received (just as no mouse or keyboard events will be received).

Try This Yourself

As with ordinary notifications, you can use nil to tell the NSDistributedNotificationCenter that you're willing to receive notifications with any name or from any app. Or both: change receiver.rb to receive every notification posted by any process on your Mac. You'll also want to change printing of notifications to something like this:

```
def next_number(notification)
  puts notification
end
```

You'll probably find that not too many distributed notifications are posted. You can provoke one by changing your desktop picture. If you look at the notification carefully, you'll see that it switches the senses of the *name* and *object* arguments. Other apps set the *object* to nil (which doesn't print), leaving you to hope that it's the only one that uses that *name*.

4.3 The App to Fenestrate

I've used the Ramaze web framework[2] to build a small and stupid web app for Fenestra to fenestrate. Here's how to use it:

1. To start the app, use this:

   ```
   $ cd counting-webapp
   $ ruby start.rb
   ```

2. To reach the app, type http://localhost:7000/ into your favorite browser's address bar.

3. The first thing you can do is create a "user" by typing a name in the text field. After the page refreshes, the new user will appear as a hyperlink.

4. If you follow the hyperlink, you'll see how often that user has been "viewed" (how often that user's page has been displayed) and how often it has been "created" (how often that particular name was typed in the text field).

5. To stop the app, just press Control-C at the command line.

2. http://ramaze.net/

4.4 Putting Notification Handling Behind the GUI

We're now ready to bolt code that receives notifications onto the back of Fenestra's GUI.

Try This Yourself

Change your version of Fenestra to receive distributed notifications from "object" *"com.exampler.counting"*. Have it receive notifications with any *name* and append them to the @log. It's fine to use to_s to render the NSNotification into a string.

You'll be launching Fenestra many times in the rest of the book, and it'd be boring to have to type "com.exampler.counting" into the text field each time. Make that string the text field's starting value. (You can do that in Interface Builder by using the inspector's Attribute tab to edit the Title field. You can do it in code by creating an outlet to the text field and setting its title attribute.)

Your same app can be used to view notifications from other sources. Use object *BackgroundChanged* to get notifications when the background picture changes. If you use Growl,[3] you can track *GrowlTicketChanged*.

My Solution

My solution is shown in Figure 4.3, on the following page.

- I've chosen to initialize the text field in awakeFromNib rather than in Interface Builder. That means I needed an outlet to the text field, and I've named it @choice. @choice is declared at ❶ and used at ❷. I used ib_outlets[4] to declare it so that Interface Builder would recognize the new outlet.
- At ❸, I set *name* to observe to nil so that I receive all notifications from app_name.
- I've hidden the work of setting ranges in two private methods: clear_log (at ❺) and record (at ❻). These two methods are closely related—for example, they both work with NSRanges and an NSTextView. They also hide those intricacies from the rest of the class. That's a clue that they really belong in a class of their own. We'll follow that clue in the next section.

3. http://growl.info/
4. Notice that you can use both ib_outlets and ib_outlet. They're aliases.

Download fenestra/simple-solution/Controller.rb

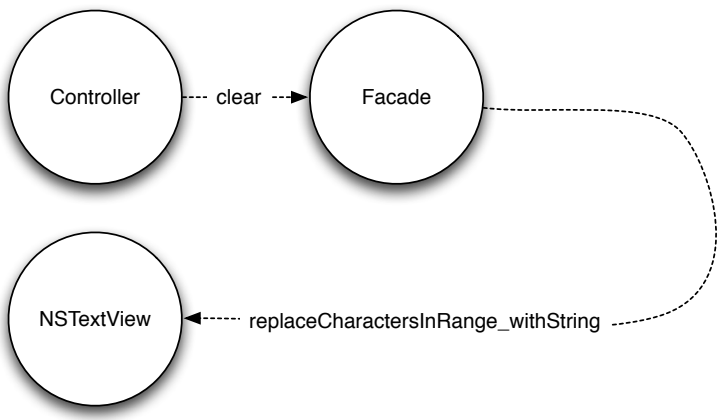
```
require 'osx/cocoa'

class Controller < OSX::NSObject
  include OSX

❶  ib_outlets :log, :logWindow, :choice
  ib_action :chooseApp

  def awakeFromNib
    clear_log
❷    @choice.stringValue = "com.exampler.counting"
  end

  def chooseApp(sender)
    app_name = sender.stringValue
    center = NSDistributedNotificationCenter.defaultCenter
    center.addObserver_selector_name_object(self, :displayNotification,
❸                                            nil, app_name)

❹    record("Observing #{app_name}...")
    @logWindow.title = app_name
  end

  def displayNotification(notification)
    record(notification.to_s)
  end

  def windowWillClose(notification)
    NSApp.terminate(self)
  end

  private

❺  def clear_log
    everything = NSRange.new(0, @log.textStorage.length)
    @log.replaceCharactersInRange_withString(everything, '')
  end

❻  def record(string)
    string += "\n"
    at_end = NSRange.new(@log.textStorage.length, 0)
    @log.replaceCharactersInRange_withString(at_end, string)
  end
end
```

fenestra/simple-solution/Controller.rb

Figure 4.3: Displaying notifications in a text view

4.5 Reopening Objective-C Classes

As of the previous section, we have two methods, clear_log and record, that are *in* Controller even though they're really *all about* NSTextView. In Java, I might create a facade class to sit between the controller and the view. It would convert those two methods into calls into an NSTextView stored in an instance variable. That would look like this:



In Ruby and Objective-C, an alternative is to *reopen* NSTextView and add behavior. As shown in Figure 4.4, on the next page, it's done the same way for an Objective-C class as for any old Ruby class: just open the class and define methods. I have changed the names: clear_log implies that an NSTextView is always a log, which is not true, and record is less descriptive than addLine.

Once the methods have been defined, some Controller code looks nicer. See ❶ in the following:

Download fenestra/reopen/Controller.rb

```
def awakeFromNib
❶   @log.clear
❷   @choice.stringValue = "com.exampler.counting"
end
```

4.6 What Now?

If you've been working along with the text, you've gotten a first experience with Interface Builder and the mechanics of coding and building a RubyCocoa app. Having put you through all that, I'm going to make you do it again. I'm not doing it (just) because I'm a sadist. I'm doing it because one of the problems a lot of people have working with a new

Download fenestra/reopen/NSPatches.rb

```
require 'osx/cocoa'

module OSX
  class NSTextView
    def clear
      everything = NSRange.new(0, textStorage.length)
      replaceCharactersInRange_withString(everything, '')
    end

    def addLine(string)
      string += "\n"
      at_end = NSRange.new(textStorage.length, 0)
      replaceCharactersInRange_withString(at_end, string)
    end
  end
end
```

fenestra/reopen/NSPatches.rb

Figure 4.4: Reopening an Objective-C class

framework and its tools is missing "muscle memory": everything you do, you have to *think about* and fumble through. That distracts and detracts from the business of learning the next feature. So, by making you reshape the application, I hope I'll get you further along the path of learning.

If you're grumbling at the thought of tool practice, rest assured there's something for you: you'll use new GUI *controls*—buttons and combo boxes—and so learn their programmatic interfaces.

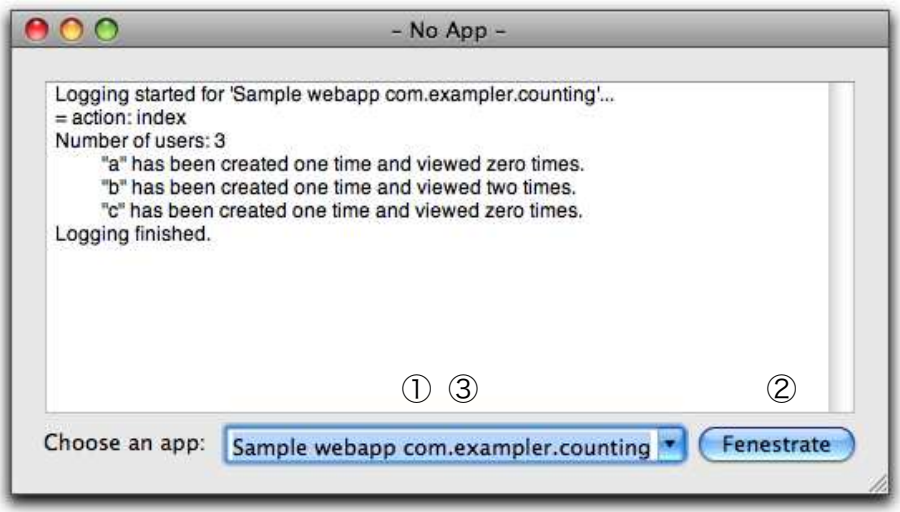
Part II

Reshaping Fenestra

Chapter 5

A Better GUI

The Fenestra interface is lousy. It's OK when all you care about is com.exampler.counting—all you have to do is strike Return. But if you wanted to work with a second app, you'd have to laboriously type in its name. This interface would be better:



- A *combo box* would let the user choose from a predefined list of known applications—but still type in whatever name she likes. Each predefined application could have its own *translator* that prints notifications in a pleasing, app-specific way. (You can see some specialized translation in the text view.)

- Pressing Return should be the same as clicking the button—so no need to fumble for a mouse. In Cocoa, that happens if a button is the *default button*. You know a button is the default button if it's colored solid blue.
- The combo box should be selected by default. Then, quick presses of the arrow keys would select another app. Alternately, you could type the app name to replace the default, with no effort needed to erase the existing text (since it's all highlighted). In either case, a simple Return would click the default button.

Starting with your existing version or the one in fenestra/reopen, change MainMenu.nib so the UI matches the earlier picture. Don't worry yet about setting the attributes of the new controls.

Here are some hints:

- Delete controls by dragging them out of the window.
- It's likely you'll need to widen your window to accommodate the combo box, which will have long strings in it.
- In the library, you can find combo boxes by searching for *combo* or *NSComboBox*. Don't use NSComboBoxCell.[1]
- There are a bewildering variety of buttons. They're all NSButtons, but they have different appearances. The version you see most often is the push button.

You can browse my solution in fenestra/reshaped-but-gutted. (It also contains UI changes from the rest of this chapter and code changes from the next chapter.)

5.1 Toggle Buttons

The first time the button is clicked, Fenestra should fenestrate (open a connection to the chosen app). The second time, let's have Fenestra "heal" the opening. That suggests the button should be a *toggle button* whose title is Fenestrate when it's toggled off and Heal when it's toggled on. You do that by setting (respectively) its Title and Alt. Title attributes, as shown on the next page.

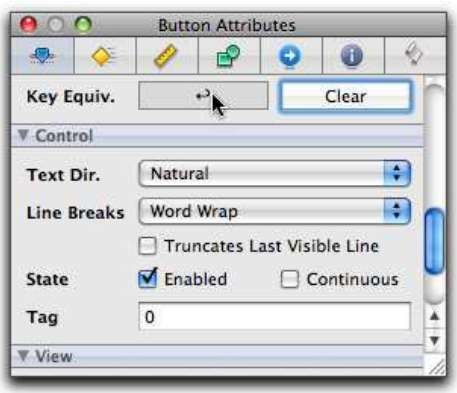
1. *Cells* are attached to controls and handle some of their work. Most often, though, you can forget they're there. For more about them, see Section 17.1, *Cells*, on page 230.

By default, buttons are just push buttons. They call an action method when clicked, but they have no notion of being on or off. You set a button to be a toggle button by changing its Mode attribute, as shown at the bottom of the previous image.

5.2 The Default Button

Recall from Section 2.3, *Menus*, on page 27 that a *key equivalent* is the generic name for keypresses that act as clicks, with Command-Q being the most familiar. In that section, you saw how to set them up programmatically, but it's easier to do it through Interface Builder. There, you set one by selecting the Key Equiv. field in the Attributes inspector and typing the key you want.

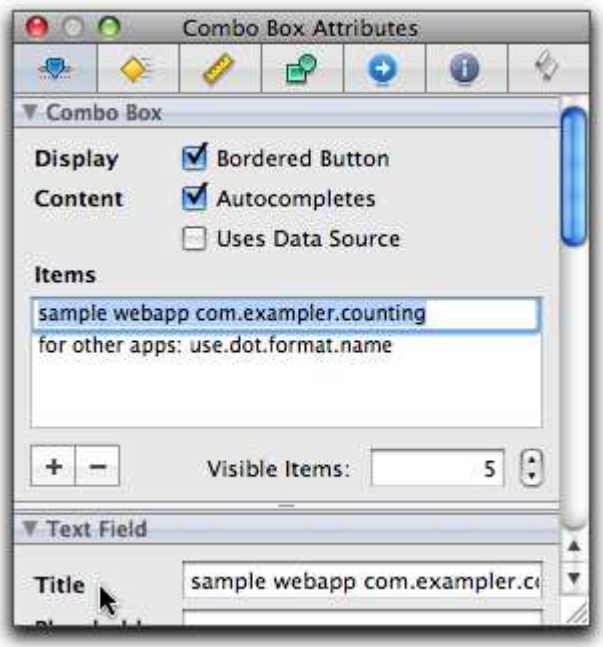
Making a button the default is layered onto this way of setting key equivalents. If you set a button's key equivalent to Return, it becomes the default button. That's what I've done in the following snapshot:



5.3 Combo Box Items

In the next chapter, we'll set the combo box items programmatically. For now, let's set them through IB. This is all done through the Attributes tab in the inspector.

A combo box combines a pop-up list with a text field. The contents of the list are set in the Items subsection of the Combo Box section, as shown here:



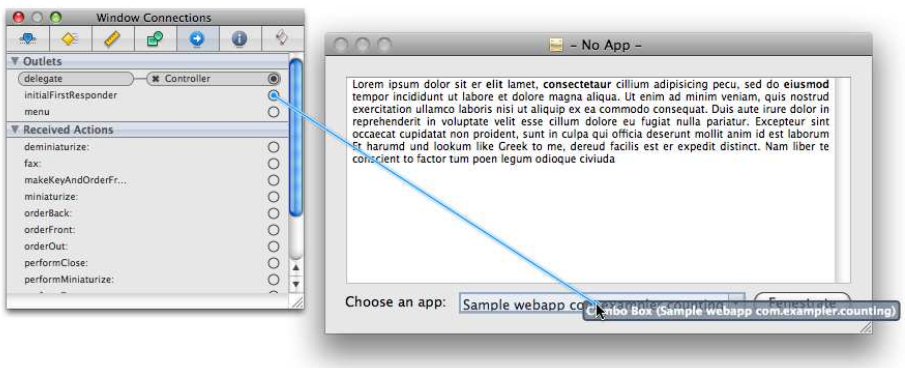
The value of the combo-box-as-text-field is set using the Title field in the Text Field section, as shown at the bottom of the earlier figure. Since I want com.exampler.counting to be the default (so I can fenestrate it by just hitting Return), I have to retype it even though it's in the list. That's annoying, which is why I want to quickly put the combo box under programmatic control.

I've also selected the Autocompletes checkbox. Typing in the text field will autocomplete any matching combo-box list items. (You can see this behavior in Safari's address bar when you start typing a URL you've visited recently.)

5.4 The Initial First Responder

Since the point of this new interface is quick fenestration and keyboard control, there's still a glitch to fix. To see it, launch the simulator and start typing. If you've been working along with the chapter (rather than using the already-completed reshaped-but-gutted version), the characters you type will appear in the text view. A user interface designer might say the text view has *focus*. A Cocoa programmer might instead say that it is the *first responder*. That's the view that gets the first chance to handle characters from the keyboard. It can refuse, in which case the Cocoa runtime looks for another view in the *responder chain*. In our case, the text view doesn't refuse, so we won't worry about the responder chain. See Apple's *Cocoa Event-Handling Guide* [App08h] for all the details.

We want the app to start out with the combo box as the first responder. That is, it should be the containing window's *initial first responder*. That's done by dragging from the window's initialFirstResponder outlet to the combo box, as shown here:



5.5 Try This Yourself

1. Hook up the controls as described in this chapter.
2. Simulate the interface.
3. Start typing. Do the characters replace the combo box text?
4. Hit Return. Does the button change to say Heal?

5.6 What Now?

We could hook up these new controls to the existing Controller. Instead, we'll create more than one controller to show how that's done.

Chapter 6

Decoupled Controllers

As this app grows, I can imagine the Controller class getting disgustingly unwieldy. My response would be to split it up into smaller controllers, each responsible for a conceptually independent part of the user interface. Come to think of it, I can already see three distinct responsibilities, so let's do the splitting now:

- The WindowController will manage the sole window, including putting the app's name in the title bar.
- The LogController will control the view used for logging events from the app. It's a simple text view, though it would probably become tree-structured in a production-ready Fenestra.
- The AppChoiceController will manage the controls that allow the user to choose an app to fenestrate and, later, heal.

The relationships between these controllers and their views are shown in the top half of Figure 6.1, on the following page. The lower objects in the picture will have a different job: translating from NSNotification objects into human-readable strings. One of them, ToString, will not be allowed to assume anything about the app that sent the notification. It will have to do the sort of raw dump we did in Chapter 4, *One Good App Observes Another*, on page 62. The other, CountingApp, will know specifically about each notification from com.exampler.counting. You won't see its code until Chapter 9, *Bundling Gems and Libraries with Your App*, on page 109.

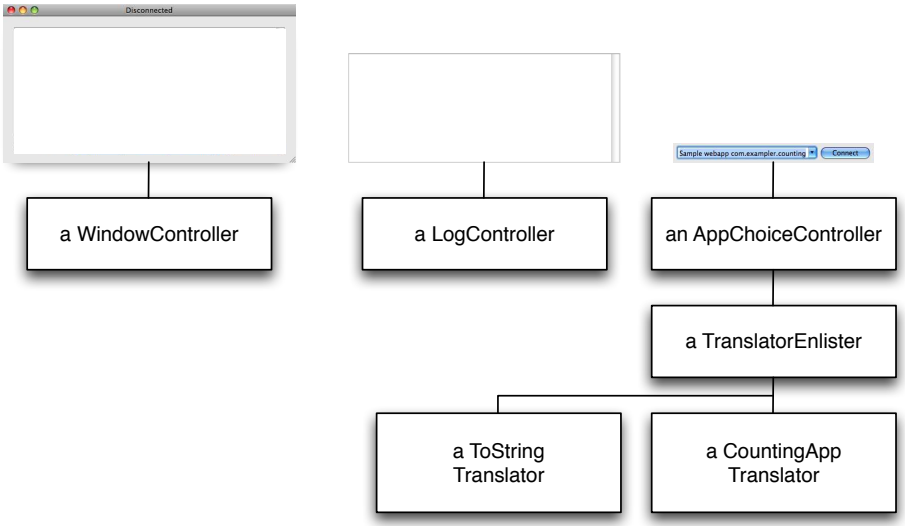


Figure 6.1: The most important objects in this version

TranslatorEnlister keeps track of translators for the rest of the application. Any class can hand it a description of an app and tell it to start up an appropriate translator. Later, TranslatorEnlister will be responsible for finding and loading user-supplied translators.

The lines in the figure represent knowledge. For example, the TranslatorEnlister has to know about its translators, but no other object does.

6.1 Ignorant Objects

There's a lot of ignorance in Figure 6.1. Although a translator produces strings that the LogController puts in the log, both the translator and LogController are ignorant of each other. By that, my exact meaning is that neither can follow a chain of instance variables (or accessor methods) to arrive at the other. Instead of the translator calling on the LogController to log a string, it will just send out an NSNotification saying "Here's a string someone might want to log." The LogController will observe such notifications and log the strings they contain.

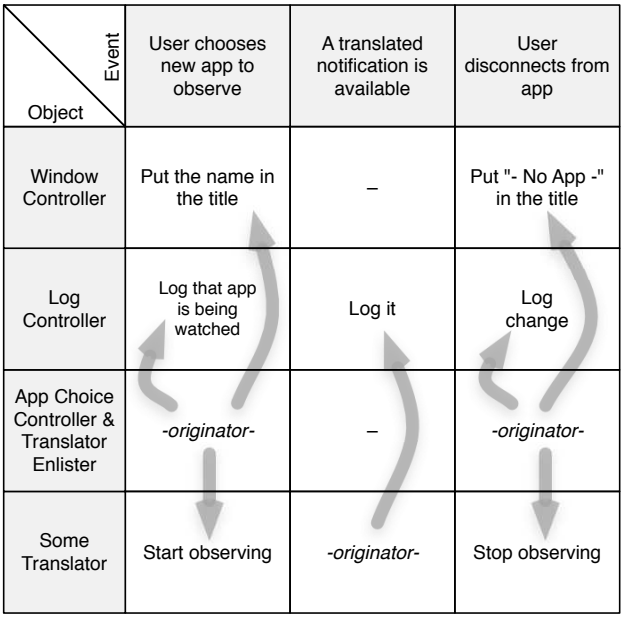


Figure 6.2: An event/object table

I learned this style of extreme decoupling from Carl Erickson and David Crosby of Atomic Object.[1] It simplifies adding new behavior. If I want some other object to do something with log messages, I don't have to change any existing code. I just have to make the new object listen for the notification.

Another reason for this style is that I'm easily overwhelmed, so I need an app structure that helps me think about everything I need to do when someone says, "Hey, I have this great new idea!" By reducing so much into objects and events that poke at them, I can use a table, like the one shown in Figure 6.2, to keep myself straight. That table isn't wildly compelling—even I can keep track of that little—but I hope you can imagine using it systematically. You add a new event and then step down the list, asking the question, "Could this object care about that event?"

1. http://www.atomicobject.com

6.2 Extracting Subclasses

Try this yourself:

1. Your current version of the app has a single Controller class. Extract the three separate classes—WindowController, LogController, and AppChoiceController—from that class, making each a subclass of Controller.

   By "extract," I mean that you should first make the new files with File > New File. Next, search through the current version of Controller to find declarations or code specific to, say, the application's window. Move that code down to the appropriate subclass (WindowController, in this case).

   You can use this template for the subclasses:

   Download fenestra/reshaped-but-gutted/controller-template

   ```
   require 'Controller'

   class SomeController < Controller
     # declare ib_outlets and ib_actions

     def awakeFromNib
       NSLog("Some controller awakes from Nib.")
     end

     # Actions, methods that respond to notifications, etc.
   end
   ```

   As you move things, delete ones no longer relevant. For example, since the text field has been replaced by a combo box, you don't need the :choice outlet. And the chooseApp method is specialized for text fields, so don't bother moving it to the AppChoiceController.

2. Add new **ib_outlets** for the new controls in this version of Fenestra. (Don't add any action methods yet.)

3. If you started this part of the book with your own code (rather than the one in fenestra/reopen), make sure you've included NSPatches.rb in your source directory. My sample solution uses it. (NSPatches.rb was described in Section 4.5, *Reopening Objective-C Classes*, on page 73.)

   Adding an existing file isn't as easy as copying it into the source folder. You must tell Xcode about it with the Project > Add to Project command. If you have trouble with that, see Section 9.1, *Manual Control*, on page 110.

4. You can see my solution in Figure 6.3, on the next page.

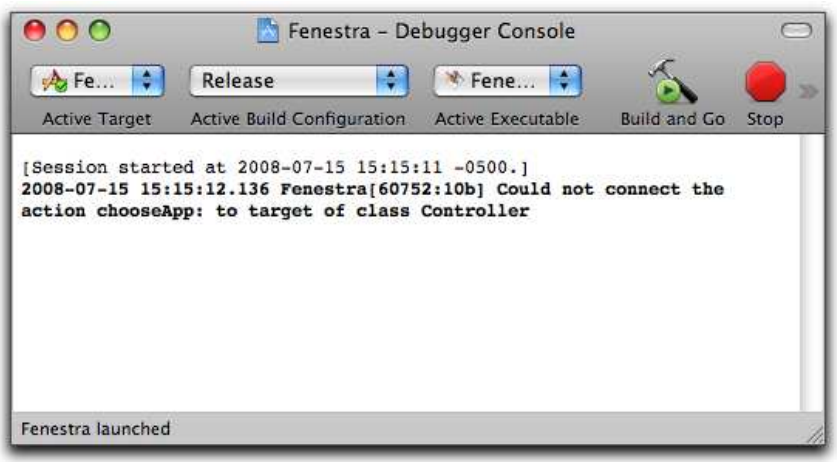
   Not much gets left behind in Controller, only an include that keeps me from having to type OSX:: in front of the NSLog calls.

   I deleted displayNotification because it was used only in the (now deleted) chooseApp. I may add it back later.

   I deleted awakeFromNib from Controller because it ended up empty.

5. Run the application. I expect you'll see something like this:

   That seems wrong. LogController is supposed to clear the log, but it hasn't. In fact, the debugger console is missing all the NSLog output, plus it has an error message:



   Why have none of the controllers been loaded? They were never declared in the nib file. Why the error message? The Controller *was* declared in the nib file, and it was never deleted, so the Cocoa

Download fenestra/reshaping-in-progress/Controller.rb

```
class Controller < OSX::NSObject
  include OSX
end
```

Download fenestra/reshaping-in-progress/WindowController.rb

```
require 'Controller'

class WindowController < Controller
  ib_outlet :logWindow

  def awakeFromNib
    NSLog("Window Choice Controller awakes from Nib.")
  end

  def windowWillClose(notification)
    NSApp.terminate(self)
  end
end
```

Download fenestra/reshaping-in-progress/LogController.rb

```
require 'Controller'

class LogController < Controller
  ib_outlets :log

  def awakeFromNib
    NSLog("Log Controller awakes from Nib.")
    @log.clear
  end
end
```

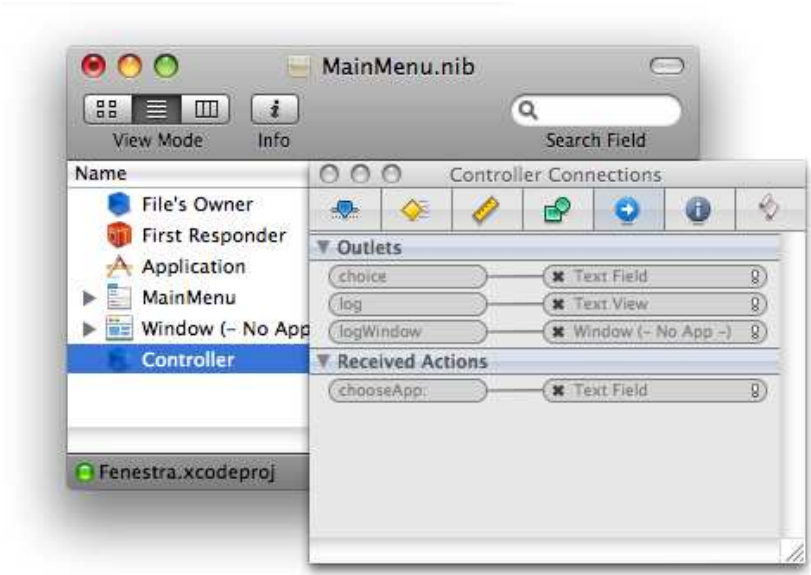
Download fenestra/reshaping-in-progress/AppChoiceController.rb

```
require 'Controller'

class AppChoiceController < Controller
  ib_outlets :comboBox, :button

  def awakeFromNib
    NSLog("App Choice Controller awakes from Nib.")
  end
end
```

fenestra/reshaping-in-progress/Controller.rb

Figure 6.3: One class split into four

runtime is still trying to establish its connections. You can see that by looking at the Controller's connections in the IB inspector:



The light gray text and stylized exclamation points tell you the other end of the connection is missing from the nib.

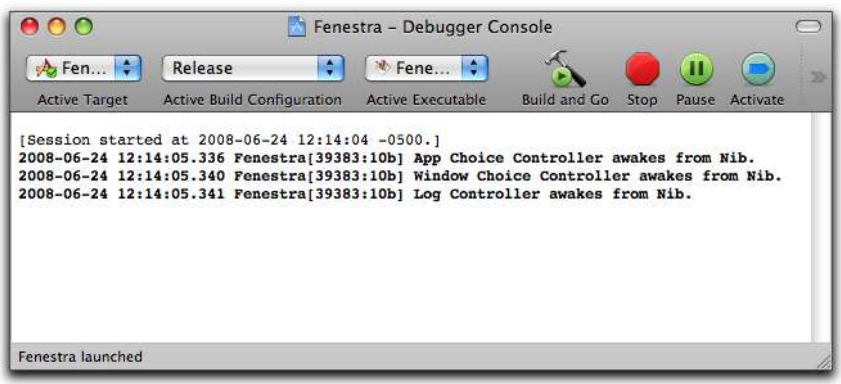
6. Get rid of the Controller in the doc window, and make it look like this:

(You create controllers in the doc window by dragging in NSObjects and changing their class on the inspector's Identity tab.)

7. Was IB clever enough to notice the class outlets? (Use the inspector's Connections tab to check.) If not, use File > Synchronize With Xcode to fix it. Then drag the outlets into place. Here's an example of the result:

Don't forget to do all three controllers.

8. If you save and then build and run, you should now see a cleared log window, and each controller should make a note in the debugger console:



You may see the log messages in a different order. There's no guaranteed order of object initialization.

9. Try closing Fenestra's window with Command-W. Unless you were more thoughtful than I was, Fenestra won't exit. Oops. Recall from Section 3.6, *Overriding Window Behavior with a Delegate*, on page 60 that the previous version of Fenestra called windowWillClose because Controller was the window's delegate. When you deleted the Controller from the doc window, the delegate connection went away. The solution is simple: make the WindowController the window's delegate.

6.3 Reacting to Button State

When a button is clicked, it invokes an action method. That action method can query the button for its state and act appropriately. Here's such code:

Download fenestra/reshaped-but-gutted/AppChoiceController.rb

```
ib_action :chooseOrHeal

def chooseOrHeal(sender)
  NSLog("AppChoiceController button pushed.")
  if @button.state == NSOnState
    NSLog("Fenestrate '#{@comboBox.stringValue}'.")
  else
    NSLog("Heal.")
  end
end
```

Before you can see that working, you'll need to connect the button to the chooseOrHeal method, using Interface Builder. (Either you can drag to the button from the AppChoiceController's chooseOrHeal received action or you can drag from the button's selector sent action to the AppChoiceController.)

For the complete description of buttons, see the NSButton class reference and *Button Programming Topics for Cocoa* [App08e].

6.4 Using Nibs to Avoid Dependencies

The AppChoiceController connected to a TranslatorEnlister is shown in Figure 6.1, on page 83. It would be easy enough for it to create that TranslatorEnlister inside its awakeFromNib:

```
@translatorEnlister = TranslatorEnlister.alloc.init
```

However, I have a learned aversion to making one class's code explicitly name another class. That tends to make the code harder to change, and it definitely makes it harder to test. Instead, I can make the connection to the TranslatorEnlister be an outlet, no different in principle from the outlets to the button and combo box. That's done at ❶, as shown here:

Download fenestra/reshaped-but-gutted/AppChoiceController.rb

```
class AppChoiceController < Controller

  # Upward to the view
  ib_outlets :comboBox, :button

  # Downward into guts
❶ ib_outlet :translatorEnlister
```

Once we tell it about TranslatorEnlister, nib loading can do the connecting for us. Having the outlet set from outside the class is a convenient form of *dependency injection*.[2]

First, we need a TranslatorEnlister to load. In this version of the application, we're just building scaffolding, so there'll be no actual translators. The TranslatorEnlister will just programmatically supply the same two bits of information we already specified in IB: what should go in the combo-box-as-a-list and what should be the initial value of the combo-box-as-a-text-field. Here's a way to do that:

Download fenestra/reshaped-but-gutted/TranslatorEnlister.rb

```
class TranslatorEnlister < OSX::NSObject
  include OSX
  attr_reader :choices, :favorite

  def init
    @favorite = "sample webapp com.exampler.counting"

    @choices = [
      @favorite,
      "for other apps: use.dot.format.name"
    ]
    super_init
  end

  def awakeFromNib
    NSLog("TranslatorEnlister awakes from Nib.")
  end
end
```

To Interface Builder, it doesn't matter in the slightest that TranslatorEnlister has nothing to do with, well, the interface. You create and connect it the same way you would any other object.

2. My favorite article on dependency injection is J. B. Rainsberger's "Injecting testability into your designs" [Rai05].

6.5 Initializing Combo Boxes

Here's how the AppChoiceController can put the information provided by the TranslatorEnlister into the combo box:

Download fenestra/reshaped-but-gutted/AppChoiceController.rb

```
def awakeFromNib
  NSLog("App Choice Controller awakes from Nib.")
❶ @comboBox.removeAllItems
  @translatorEnlister.choices.each do | t |
❷   @comboBox.addItemWithObjectValue(t)
  end
❸ @comboBox.stringValue = @translatorEnlister.favorite
end
```

❶ We've already initialized the list to have two items in Interface Builder. We could remove them there, but it's prudent to clear the list anyway. Remove this line to see a list with duplicates.

❷ This is where items are added to the list. The method name, addItemWithObjectValue, hints that the argument can be something other than a string. Indeed, it can be any object. Try changing the choices array to be an array of integers. You'll see that they display reasonably, and they're correctly logged in chooseOrHeal when the button is clicked.

❸ This line sets the value of the text field. It does nothing to the combo box's list. An alternate way to get the same effect would be to use selectItemAtIndex with the argument 0.

See Apple's *Combo Box Programming Topics* [App08i] for more on combo boxes.

6.6 What Now?

We now have four objects (three controllers and a TranslatorEnlister) that, for the most part, have no references to each other—but they have to exchange information. I'm going to use the notification system (as described in Section 4.1, *Notifications Within an App*, on page 62) to do that. That's reasonably straightforward: every arrow in Figure 6.2, on page 84, turns into the posting of a notification. I'll briefly show what that code looks like in the next chapter. My ulterior motive for doing that is to motivate the chapter after that, which uses Ruby to make such code more pleasant.

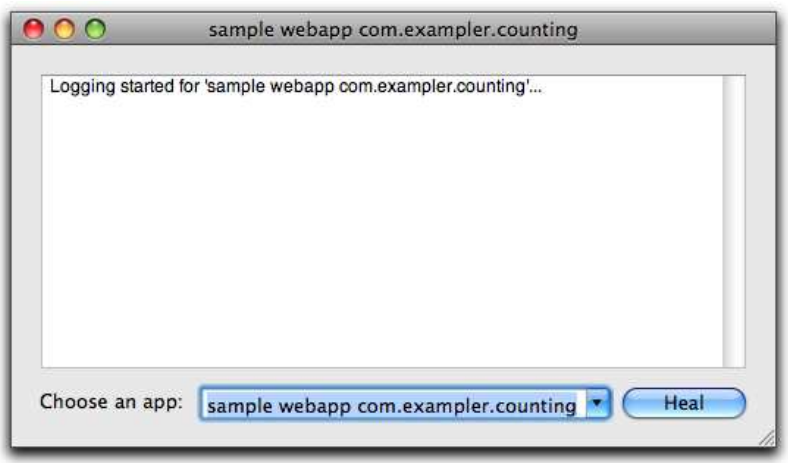
Chapter 7

Notifications Connect Decoupled Objects

Fenestra is now nicely decoupled, but it's time for it to *do* something. In this chapter, the different objects will communicate via *notifications*. I don't think you'd learn enough by typing in the required code, so I recommend you just read until Section 7.2, *Try This Yourself*, on page 96.

7.1 Controllers

fenestra/reshaped-with-notifications is a version of Fenestra that uses notifications. I'll illustrate the control flow by showing what happens between the moment a user clicks the Fenestrate button and the moment this text appears in the log window:



The first method called is in AppChoiceController:

Download fenestra/reshaped-with-notifications/AppChoiceController.rb

```
def chooseOrHeal(sender)
  if @button.state == NSOnState
❶   @last_choice = @comboBox.stringValue
❷   Center.postNotificationName_object_userInfo(AppChosen,
                                                self,
                                                :app_name => @last_choice)
  else
❸   Center.postNotificationName_object_userInfo(TimeToForgetApp,
                                                self,
                                                :app_name => @last_choice)
  end
end
```

❶ The user's choice is fetched from the combo box. It's stored in an instance variable so that it can also be displayed in a "Logging stopped" message (sent at ❸). I can't just ask the combo box again because there's no guarantee the user hasn't messed with the combo box before clicking Heal.

❷ Here, a notification is posted to the default NSNotificationCenter. (I aliased that to Center in the Controller superclass.) I used the constant AppChosen instead of a literal string because typos in constant names fail more obviously. I've defined it and related constants in a new file, Constants.rb.

Back when the LogController awoke from its nib, it told the default notification center it was interested in the AppChosen notification (no matter what object posted it). That's shown at ❶ here:

Download fenestra/reshaped-with-notifications/LogController.rb

```
def awakeFromNib
  @log.clear

❶   Center.addObserver_selector_name_object(self, :note_choice,
                                            AppChosen, nil)
  Center.addObserver_selector_name_object(self, :forget_app,
                                          TimeToForgetApp, nil)
  Center.addObserver_selector_name_object(self, :log_app_fact,
                                          AppFactAvailable, nil)
end
```

Therefore, when the AppChoiceController posts the AppChosen notification, the NSNotificationCenter calls LogController's note_choice method, which logs the message we want:

Download fenestra/reshaped-with-notifications/LogController.rb

```
def note_choice(notification)
  @log.addLine("Logging started for '#{notification.userInfo[:app_name]}'...")
end
```

You can check that the other behaviors, as shown in Figure 6.2, on page 84, are implemented. For example, look in WindowController to see how it handles the AppChosen notification.

7.2 Translators and the Rising Tide of Ugliness

Meanwhile, the TranslatorEnlister is listening to see whether anyone has chosen an app to fenestrate. When it receives an AppChosen notification, it looks up a translator and starts it working. For the moment, it knows of only one translator, Translators::ToString, which produces the same log messages that the previous version of Fenestra gave.

ToString

ToString, shown in Figure 7.1, on the following page, is fairly simple.

❶ The initialization method just remembers what app this ToString instance is listening for.

❷ The ToString instance listens for *distributed* notifications from the chosen app.

❸ When it receives a distributed notification, it translates it into a string and then posts that string for whatever objects might be interested in it (in this case, the LogController).

Try This Yourself

If you start Fenestra and the counting web app,[1] then have Fenestra fenestrate the counting web app, then heal the fenestra, and *then* do something with the web app (such as refresh a page), you will notice that ToString is still happily receiving distributed notifications from com.exampler.counting and still happily posting translations, which LogWindow happily prints.

1. You can find instructions about starting the web app in Section 4.3, *The App to Fenestrate*, on page 70.

Download fenestra/reshaped-with-notifications/ToString.rb

```
module Translators
  class ToString < OSX::NSObject
    include OSX
    include Announcements

❶   def initForApp(app)
      @app = app
      init
    end

❷   def listen
      center = NSDistributedNotificationCenter.defaultCenter
      center.addObserver_selector_name_object(self, :translate,
                                              nil, @app)
    end

❸   def translate(notification)
      center = NSNotificationCenter.defaultCenter
      center.postNotificationName_object_userInfo(
                  AppFactAvailable, self,
                  :message => notification.to_s)
    end

  end
end
```

fenestra/reshaped-with-notifications/ToString.rb

Figure 7.1: ToString

So, healing doesn't actually work. Make it work by making ToString remove itself as an observer from the NSDistributedNotificationCenter when it receives a *local* notification named by the constant TimeToForgetApp.

My solution is shown in Figure 7.2, on the following page.

❶ Since more than one method will use each of the notification centers, I name them with instance variables.

❷ Here, ToString observes TimeToForgetApp notifications.

❸ Here, both notification centers are told to forget about the ToString. In my first version, I didn't have the local notification center forget the ToString. What would have been the consequences? Each click of the Heal button would have created a "zombie" ToString that wouldn't listen to remote apps (and so wouldn't clutter the log) but

Download fenestra/reshaped-with-notifications/ToString.resign.rb

```
def initForApp(app)
  @app = app
❶ @remote_center = NSDistributedNotificationCenter.defaultCenter
  @center = NSNotificationCenter.defaultCenter
  init
end

def listen
  @remote_center.addObserver_selector_name_object(self, :translate,
                                                  nil, @app)

❷ @center.addObserver_selector_name_object(self, :forget_app,
                                           TimeToForgetApp, nil)
end

❸ def forget_app(notification)
  @center.removeObserver(self)
  @remote_center.removeObserver(self)
end
```

fenestra/reshaped-with-notifications/ToString.resign.rb

Figure 7.2: A ToString that forgets

would listen for TimeToForget notifications, so it would pointlessly ask the @remote_center to remove it. Since there's no effect from removing an observer that's already been removed, all that is only trivially wasteful. It would take a lot of fenestrations to have any noticeable effect, but at the last minute I thought of what Al Gore would say and decided to completely clean up after myself.

Because removeObserver is documented for NSNotificationCenter and not for its subclass NSDistributedNotificationCenter, you might have used removeObserver_name_object instead. They have the same effect if you give nil values for the *name* and *object* arguments. I'd say the Cocoa documentation for subclasses is better than average at clearly directing your attention to superclass methods, but it's still prudent to look up the inheritance chain for the method that's perfect for what you want to do.

7.3 What Now?

The previous section was titled "Translators and the Rising Tide of Ugliness." The rising tide refers to all the calls to addObserver_selector_name_object and postNotificationName_object_userInfo. Part of the problem is the awkwardness of Cocoa method names *when translated into Ruby*. I am a fan of Objective-C-style keyword arguments, but their virtues are revealed only when the keywords and arguments interleave as intended. Another problem is that these two methods are more general than we need, so we have to keep providing information that's not important for our purposes.

As is so often the case, the solution to ugliness is to look for duplication and remove it. In the next chapter, we're going to change this chapter's code to use an aggressively Ruby-like interface to notifications. Although that has nothing to do with RubyCocoa per se, I believe happy RubyCocoa programming—and happy programming in general—happens when you program to the interface *you wish you had* and then adapt your interface to the one the system provides.[2]

2. I learned this from Steve Freeman.

Chapter 8

More Expressive Code

Lisp programmers are notorious for solving problems by first inventing a "little language" that makes the solution easy to express and then writing the solution in that language. Many Ruby programmers have the same habit, but the term we use is more often *domain-specific language* (DSL). DSLs are built by defining clever methods. Here's an example that's built right into Ruby:

```
attr_accessor :var
```

This DSL method solves the "I'm tired of writing getters and setters" problem by giving us a way to declare what we want and having attr_accessor write the right methods for us.

The line marked ❶ shows a use of a different DSL method, one built into RubyCocoa:

Download fenestra/reshaped-with-dsl/AppChoiceController.rb

```
❶ ib_action :chooseOrHeal do | sender |
    if @button.state == NSOnState
      @last_choice = @comboBox.stringValue
      post(AppChosen, :app_name => @last_choice)
    else
      post(TimeToForgetApp, :app_name => @last_choice)
    end
  end
```

That line means the same thing as these two lines from the previous version of Fenestra:

Download fenestra/reshaped-with-notifications/AppChoiceController.rb

```
ib_action :chooseOrHeal
def chooseOrHeal(sender)
```

This previously undescribed form of ib_action can simultaneously declare an action to Interface Builder and define a method named for that action. That's not a huge win, but it eliminates a redundant use of a name—and that's almost always a good thing.

Here's a similar method of my own:

Download fenestra/reshaped-with-dsl/WindowController.rb

```
delegate_method :windowWillClose do | notification |
  NSApp.terminate(self)
end
```

If I were talking to you about this class and mentioned windowWillClose, I'd probably say, "The delegate method windowWillClose" rather than just "The method windowWillClose." If that kind of reminder is useful in speech, it's probably also useful in code.

8.1 A DSL for Notifications

I've extended RubyCocoa with my own small DSL for notifications. In Figure 8.1, on the next page, we can see how notifications were handled in the previous version of WindowController vs. how we'll handle them from now on:

- The major change here is that registering a method with the default NSNotificationCenter can be folded into the declaration of that method.
- A secondary change is that I got tired of typing notification.userInfo[], so I defined the [] method on NSNotification.

How does the NSNotificationCenter class find out what observer methods to call? As it defines each method, on_local_notification stashes its name somewhere. Then connect_all_notification_observers uses the stash to tell the NSNotificationCenter about all the observers at once. In our case, connect_all_notification_observers is called in the abstract Controller class's awakeFromNib method:

Download fenestra/reshaped-with-dsl/Controller.rb

```
def awakeFromNib
  connect_all_notification_observers
  # ...
end
```

Individual controllers don't have to worry about wiring objects together.

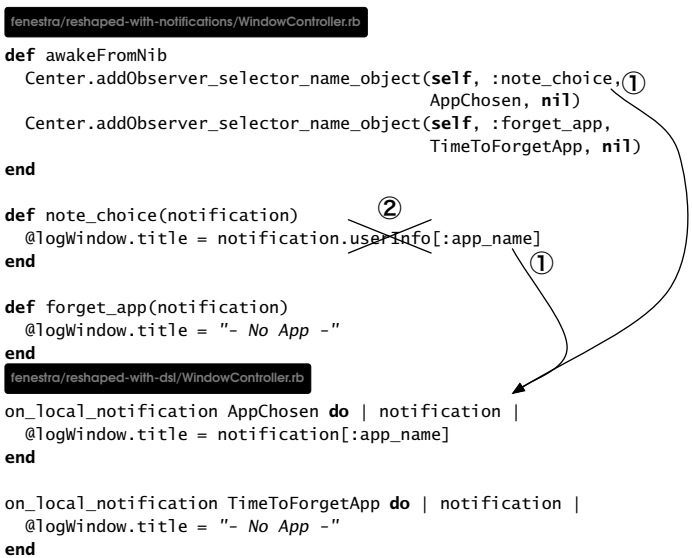


Figure 8.1: Receiving notifications the new way

Download fenestra/reshaped-with-dsl/WindowController.rb

```
class WindowController < Controller
  ib_outlet :logWindow

  on_local_notification AppChosen do | notification |
    @logWindow.title = notification[:app_name]
  end

  on_local_notification TimeToForgetApp do | notification |
    @logWindow.title = "- No App -"
  end

  delegate_method :windowWillClose do | notification |
    NSApp.terminate(self)
  end
end
```

fenestra/reshaped-with-dsl/WindowController.rb

Figure 8.2: DSLification!

I like method-defining methods such as ib_action, delegate_method, and on_local_notification because they're a quick visual hint about what role a method plays in the interconnected web of objects and methods that makes up an app. Refer to the class in Figure 8.2, on the preceding page, for instance; it seems more expressive than it used to be.

8.2 RubyCocoa Has Two Ways of Referring to Superclasses

If a Controller subclass needs to do anything special upon awakening from the nib, it should remember to call the superclass. Here's the awakeFromNib from LogController:

Download fenestra/reshaped-with-dsl/LogController.rb

```
def awakeFromNib
  @log.clear
  super
end
```

The first time I typed that method, I used super_awakeFromNib instead of super. It was reflexive: LogController is some kind of NSObject, and you call overridden Cocoa methods by prepending super_ to their name. Right?

Well, not exactly. The phrase "Cocoa methods" is sloppily vague, and I paid the usual price for sloppiness. A method super_foo calls the method foo in the first *Objective-C* ancestor class. The pseudomethod super calls the first *Ruby* ancestor. You can see the difference by running the example in Figure 8.3, on the next page. You'll see this output:

```
$ ruby super-and-super.rb
super_description says: <RubyFromRuby: 0x374a50>
but super says: Some kind of RubyFromObjC
```

Since the awakeFromNib I wanted is in the Ruby class Controller, I needed to call super.

8.3 Shorthand for Posting Notifications

I got pretty tired of using self as the *object* argument to postNotification-Name_object_userInfo, so I decided on a way to avoid that. It starts with a class wrapped around NSNotificationCenter.defaultCenter:

Download fenestra/reshaped-with-dsl/Controller.rb

```
def awakeFromNib
  # ...
  @outbox = NotificationOutBox.new(:local, :object => self)
end
```

Download rubycocoa-oddities/super-and-super.rb

```
require 'osx/cocoa'

# This file shows that the RubyCocoa super_x way of calling a
# superclass method ONLY looks in an Objective-C object, not a
# superclass that's defined in Ruby.

class RubyFromObjC < OSX::NSObject
  def description
    "Some kind of RubyFromObjC"
  end
end

class RubyFromRuby < RubyFromObjC
  def description
    "super_description says: " + super_description +
      "\nbut super says: " + super +
      "\nThe super_description comes from NSObject#description."
  end
end

if $0 == __FILE__
  puts RubyFromRuby.alloc.init.description
end
```

rubycocoa-oddities/super-and-super.rb

Figure 8.3: Two kinds of superclass

A NotificationOutBox (and its partner, NotificationInBox) remembers particular values for you and uses them unless you tell it differently. Here's how you post a notification using it:

```
@outbox.post(AppChosen, :app_name => @last_choice)
```

The hash argument is used to fill in the *userInfo*.

I hope you find that method call more appealing than this one:

```
NSNotificationCenter.defaultCenter.postNotificationName_object_userIn↩
fo(AppChosen, self, :app_name => @last_choice)
```

But even that's too much typing, so I've also defined post in Controller so that controller code needn't bother with @outbox. You can see the result in Figure 8.4, on the following page.

fenestra/reshaped-with-notifications/AppChoiceController.rb

```
def chooseOrHeal(sender)
  if @button.state == NSOnState
    @last_choice = @comboBox.stringValue
    Center.postNotificationName_object_userInfo(AppChosen,
                                                self,
                                                :app_name => @last_choice)
  else
    Center.postNotificationName_object_userInfo(TimeToForgetApp,
                                                self,
                                                :app_name => @last_choice)
  end
end
```

fenestra/reshaped-with-dsl/AppChoiceController.rb

```
ib_action :chooseOrHeal do | sender |
  if @button.state == NSOnState
    @last_choice = @comboBox.stringValue
    post(AppChosen, :app_name => @last_choice)
  else
    post(TimeToForgetApp, :app_name => @last_choice)
  end
end
```

Figure 8.4: Posting notifications the new way

8.4 Try This Yourself

Update TranslatorEnlister to use the new DSL. You'll use the instance methods defined in module Notifiable, which is contained in Notifiable.rb.

My solution is in Figure 8.5, on the next page. The changes are marked in the margins. I hope they don't need explanation.

DSLs in Context

DSLs are, at least to some extent, a matter of taste. In your day-to-day programming, you might not go as wild with them as I do, but you'll have to put up with mine for the rest of the book.

I should note that I didn't try to make this NSNotificationCenter-hiding DSL broadly useful. It does what I need for Fenestra. If you want to reuse it, great! But you should expect to tailor it, starting with the tests in notifications/test.

Download fenestra/reshaped-with-dsl/TranslatorEnlister.dsl.rb

```
require 'Notifiable'

class TranslatorEnlister < OSX::NSObject
  include OSX
  include Announcements
❶ include Notifiable

  attr_reader :favorite

  def init
    @favorite = "sample webapp com.exampler.counting"

    k = Struct.new(:display_name, :app_name, :template)
    @translators = [
      k.new(@favorite, "com.exampler.counting", Translators::ToString),
    ]

❷   connect_all_notification_observers

    super_init
  end

  def choices
    @translators.collect { | t | t.display_name } +
      [ "for other apps: use.dot.format.name" ]
  end

❸ on_local_notification AppChosen do | notification |
❹   display_choice = notification[:app_name].to_ruby
    translator = @translators.find { | t |  t.display_name == display_choice }
    if translator
      translator.template.alloc.initForApp(translator.app_name).listen
    else
      Translators::ToString.alloc.initForApp(display_choice).listen
    end
  end
end
```

fenestra/reshaped-with-dsl/TranslatorEnlister.dsl.rb

Figure 8.5: Using the DSL with TranslatorEnlister

That lack of ambition was not (only) laziness. I have enough well-earned humility to know that if I try to anticipate your needs, I'll fail and produce a bad DSL. I'm also a fan of Michael Feather's notion of intentionally "stunting a framework"—that is, of helping someone understand a framework by forcing her to change it.[1]

8.5 What Now?

The ToString translator needs to be updated to use the DSL to receive both within-app and between-app notifications. Although I could cover that now, it wouldn't teach you much about RubyCocoa or Cocoa. I'd rather defer it until Chapter 11, *Persistent User Preferences*, on page 127, when you'll be using Cocoa bindings, defaults, and preferences to describe and load custom translators.

Instead, I want to shift to hard-core pragmatics: how to set up your RubyCocoa apps so that you can edit, build, and distribute them with ease.

1. See http://www.artima.com/weblogs/viewpost.jsp?thread=8826.

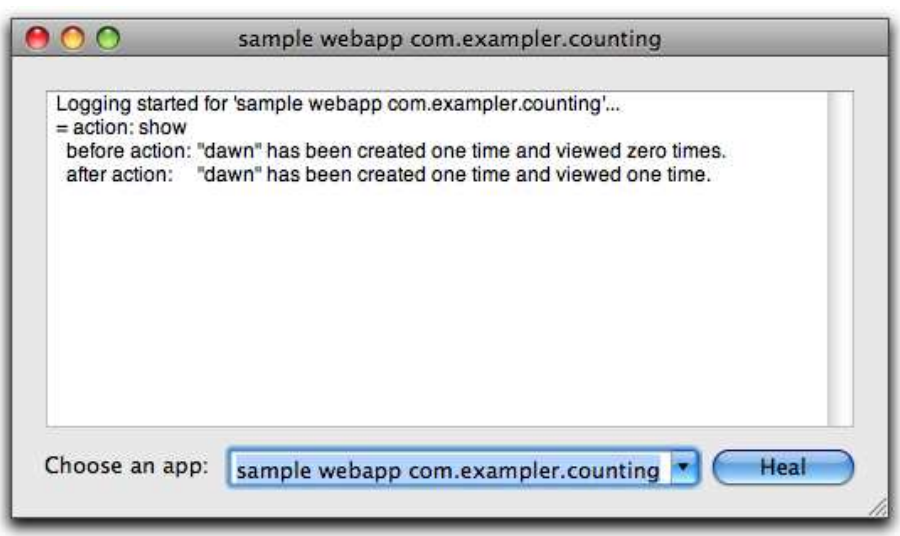
Part III

Project Mechanics

Chapter 9

Bundling Gems and Libraries with Your App

In any decent-sized software project, a certain amount of bookkeeping and organization work is required before the *real* work can run smoothly. This chapter and the next help you with that organization. To give us something to organize, I've written a translator tailored to com.exampler.counting. It produces output like this:



For this chapter, what matters are the lines containing numbers. I got fancy and turned numbers into words—"one" instead of "1"—and correctly pluralized the nouns the numbers modify. That is, you see "one time" instead of "one times," as is so depressingly common.

That's actually a bad UI for logging output, since it makes the log harder to skim, but it gives me an excuse to use the Linguistics gem. That gem works like this:

```
irb(main):001:0> require 'Linguistics'
=> true
irb(main):002:0> Linguistics.use(:en)
=> [String, Numeric, Array]
irb(main):003:0> [1.en.numwords, 2.en.numwords]
=> ["one", "two"]
irb(main):004:0> ['time'.en.plural(1), 'box'.en.plural(2)]
=> ["time", "boxes"]
```

If I want other people to use Fenestra, I should not assume they have the Linguistics gem installed in /usr/lib/ruby/user-gems or that Fenestra is so wonderful they'll happily take the time to install a gem or that they'd be happy with an app that requires running an installer (instead of the usual "drag this into Applications" installation method). After this chapter, you'll have an app that

- Has its own private folder for gems.
- Also has a private folder for libraries whose authors haven't bothered to make them into gems.
- Will *fail* if you forget to put a non-Apple gem or library in its private folder. That avoids the embarrassment of having your app fail only when you copy it to someone else's machine and say, "Watch this!" Then you double-click the app.[1]

9.1 Manual Control

As soon as I decide to use a gem or library, I put it into the app bundle. To make sure that I don't forget, I always run the app with a load path that excludes site-specific libraries and gems. This section is about how that's done. There's a more automated alternative that's unfortunately riskier; see Section 9.2, *Standaloneify*, on page 114.

1. You'll have to put the gem into the private folder even if you never plan to use the app on any other machine.

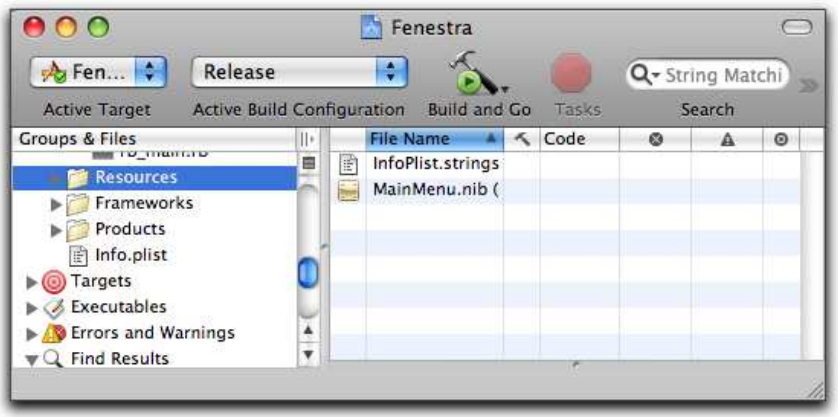
Preparing a Directory

Follow these steps to prepare a directory:

1. Use the Finder or Terminal to create a third-party folder inside your app's top-level directory:

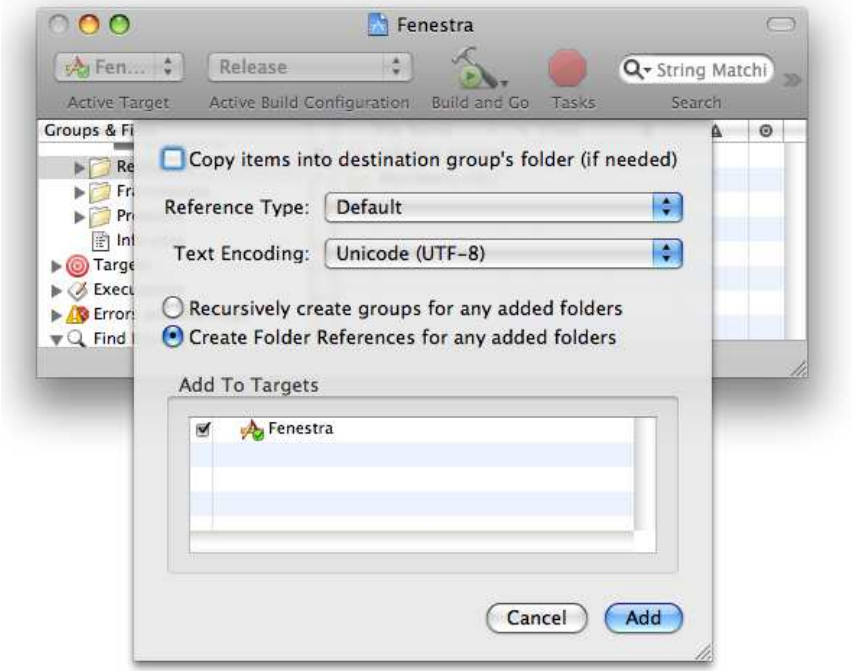
```
$ cd fenestra/bundling
$ mkdir third-party
```

2. Back in Xcode, decide where you want the third-party directory to appear in the project window. In the following example, I'm putting it in Resources:



Where you put it is unimportant for running the app—all Ruby files end up in the application bundle's Contents/Resources folder.

3. Use Project > Add to Project to add third-party. After you pick the directory, you'll be shown this dialog box:



Because you'll be creating and managing a hierarchy of third-party gems and libraries, choose Create Folder References. (It's probably not selected by default.) If you choose Recursively Create Groups, the on-disk folder structure will be flattened in the application bundle.

4. The folder project-files (part of this book's code download) contains the skeleton of an app that follows the suggestions in this chapter. From it, copy path-setting.rb into your source folder. Add it to the project by selecting Project > Add to Project. Read on for an explanation of what path-setting.rb does.

Updating rb_main.rb

path-setting.rb updates gem and library search paths so that the following items are true:

- A gem placed within third-party/gems can be found by the app. The same should be true of a nongem library in third-party/lib.

Download fenestra/bundling/rb_main.rb

```
require 'osx/cocoa'
❶ require 'path-setting'

def rb_main_init
  path = OSX::NSBundle.mainBundle.resourcePath.fileSystemRepresentation
  rbfiles = Dir.entries(path).select {|x| /\.rb\z/ =~ x}
  rbfiles -= [ File.basename(__FILE__) ]
  rbfiles.each do |path|
    require( File.basename(path) )
  end
end

if $0 == __FILE__ then
  OSX::NSLog "RubyCocoa version is #{OSX::RUBYCOCOA_VERSION}."
❷   RubyCocoaLocations.restrict_load_path_to_OSX_defaults
❸   RubyCocoaLocations.make_chosen_libs_and_gems_available
  rb_main_init
  OSX.NSApplicationMain(0, nil)
end
```

fenestra/bundling/rb_main.rb

Figure 9.1: Code to isolate an app at all times

- A gem in /usr/lib/ruby/user-gems will *not* be found.
- A gem in /usr/lib/ruby/gems can still be found. These are the gems delivered with Leopard, so it's safe to assume they exist on your users' machines.
- A nongem library in /usr/lib/ruby/site_ruby or a directory named in your RUBYLIB environment variable will *not* be found.

To use path-setting.rb, add to rb_main.rb the highlighted lines in Figure 9.1.

Installing a Gem

Install a gem in the third-party folder using a command like this:

```
$ gem install Linguistics --no-rdoc --no-ri --remote --install-dir ↩
third-party/gems
```

Here's an important fact: Xcode treats the third-party folder as a unit. It knows to update the copy in the build/Release folder only if the timestamp of the third-party folder itself changes. That doesn't happen when you add a gem to the third-party/gems subfolder. So to get the new gem

Renaming Ruby Files

Third-party code isn't the only time you'll need to manually clean the build. Try this yourself: use Xcode to rename TranslatorEnlister.rb to TranslatorFinder.rb. Build a new version. Now look inside the build directory. As of Xcode 3.0, this is what you'll see:

```
$ ls build/Release/Fenestra.app/Contents/Resources/Transl*
build/Release/Fenestra.app/Contents/Resources/TranslatorEnlister.rb
build/Release/Fenestra.app/Contents/Resources/TranslatorFinder.rb
```

The new file was copied in, but the old file is still there. Since rb_main.rb loads all the files in Resources when it starts, that could lead to some nasty surprises. When in doubt, select Build > Clean.

into the build, clean the build with Build > Clean before using Build > Build and Go.

Installing a Library

When a library isn't bundled into a gem, put it into lib. By way of example, suppose the notifications DSL was an old-fashioned library. It would be installed like this:

```
$ cd notifications
$ ruby setup.rb config --siterubyver=../fenestra/bundling/third-party/lib
---> lib
<--- lib
$ ruby setup.rb install
rm -f InstalledFiles
---> lib
mkdir -p /Users/marick/.../fenestra/bundling/third-party/lib
install Notifiable.rb /Users/marick/.../fenestra/bundling/third-party/lib/
install NotificationBox.rb /Users/marick/.../bundling/third-party/lib/
<--- lib
$
```

Again, you'd have to clean the build (Clean > Build in Xcode) before the new files will make it into the built app.

9.2 Standaloneify

standaloneify is a tool that comes with RubyCocoa. It runs your app, tracks what it requires, and writes a new app that copies the required

libraries and gems into the app's bundle and sets up the app's load path so that it can find them there.

To use standaloneify, you need to put it in your shell's load path:

```
PATH=$PATH:/System/Library/Frameworks/RubyCocoa.framework/Versions/Cu ↩
rrent/Tools
```

You run it like this:

```
$ cd build/Release
$ ruby -S standaloneify.rb -d /tmp/Fenestra.app Fenestra.app
```

I've tried that with two versions of Fenestra. For the one in fenestra/text-field-to-view, all went well: the stand-alone application launched and appeared to work. The one in fenestra/reshaped-dsl did not fare so well. It didn't launch at all. For some reason, standaloneify copied the RubyCocoa framework itself into the app bundle but didn't adjust the load path to include the stand-alone copy. The result is this message on the console:

```
Jul  2 12:48:37 frex [0x0-0x3ce3ce].com.apple.rubycocoa.FenestraApp[ ↩
19800]: /Users/marick/tmp/Fenestra.app/Contents/Resources/rb_main.rb ↩
:25:in `require': no such file to load -- osx/cocoa (LoadError)
```

I haven't figured out why. I think it's a bug. Once the madness of book writing is over, I'll investigate further and file a bug report if needed.

Whatever the reason for that failure, there's an important and different reason standaloneify can fail. Suppose your app—*or any code it requires*—only conditionally requires a library. Consider this code:

```
ib_action :chooseFancyOutput do | sender |
  require 'Linguistics'
  ...
end
```

Since the app doesn't load Linguistics without user intervention, standaloneify doesn't see that require executes, so Linguistics won't end up in the app bundle.

That's a much worse problem than the one I saw with the DSLified version of Fenestra. At least that one failed right away. In *this* hypothetical case, you might think the app works until some user decided on fancy output.

Because of that risk, standaloneify is for the bold (including people who want to distribute apps for pre-Leopard versions of OS X, which the technique in Section 9.1, *Manual Control*, on page 110, does not support).

9.3 What Now?

Now that you have one folder full of code, you might start wanting to make others. I myself am itching to have util and translators folders. The next chapter shows you how to organize RubyCocoa projects hierarchically. It's not as obvious as you might think, and the way you go about it depends on whether you want to edit Ruby code with Xcode or some other editor.

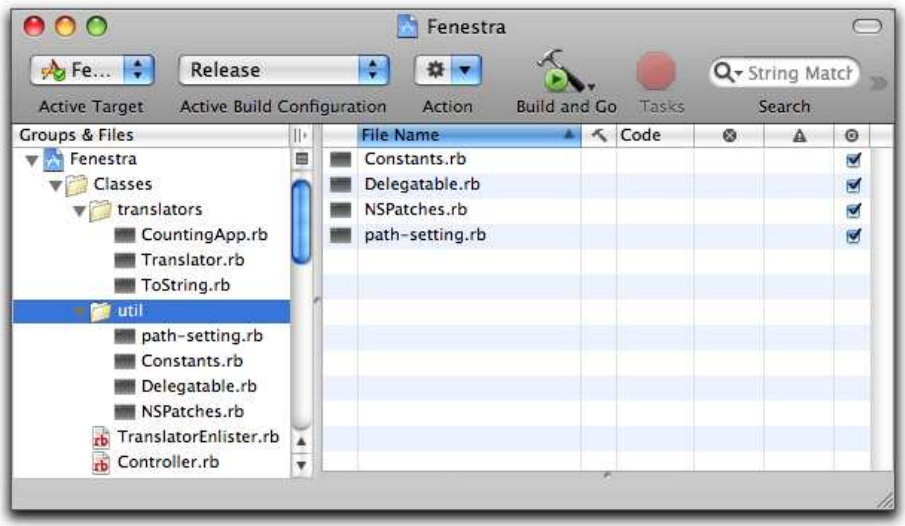
Chapter 10

Project Organization, Builds, and Your Favorite Editor

I remember the thrill, back around 1980, when I first worked on an operating system that let me have directories *within* directories. I got rather attached to the idea. Unfortunately, for the obsessively hierarchical, Xcode would prefer that all your Ruby files live directly in the Resources folder of the app bundle, without any of them in any subfolders. This chapter is for people who object to that. It's also for people who want to use rakefiles for building the app (probably because they have a favorite Ruby editor that's not Xcode).

10.1 Groups

The most recent version of Fenestra has the utility file NSPatches.rb in the same folder as the translator ToString.rb and the controller AppChoiceController.rb. Xcode's internal display can (mostly) hide that truth from you. I've arranged for Xcode to show me the Fenestra files like this:

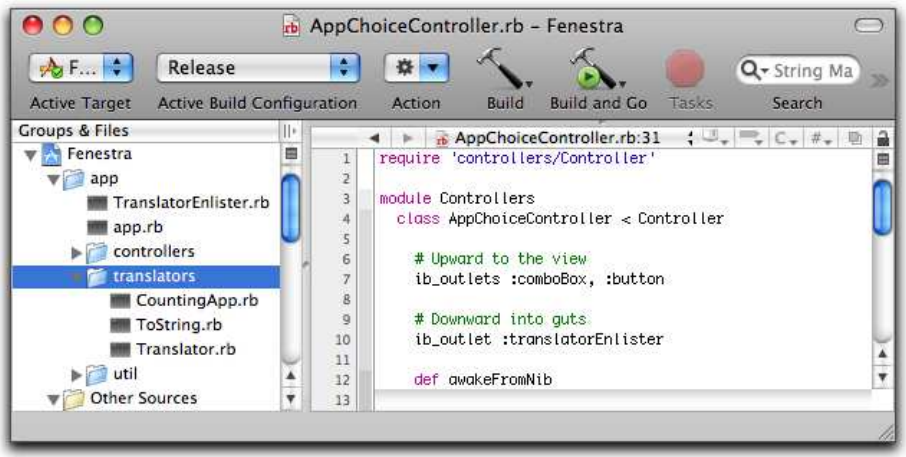


Despite being in the same folder on disk, Xcode shows the utility files in a different *group* than the translators. You could create a subgroup of util by selecting it and choosing Project > New Group.

Groups have most of the advantages of folders, but you can't have files with the same name in two different groups.

10.2 Using Xcode with Hierarchical Project Folders

Since Xcode groups are invisible outside Xcode, you might want to have Xcode work with a folder hierarchy on disk. Folders look just like groups, but they're blue instead of yellow. If you're reading the PDF version of the book, you can see that here:



In version 3.0, Xcode doesn't really understand folders. By that, I mean this:

- You can't create folders from within Xcode. You have to create them with the Finder or command line and then add them to the project.
- Although you can expand folders to view them hierarchically, you can't rename or move files.
- You can edit files, but Xcode won't consistently save them before building, even if you've told it to do so.[1] I've written an AppleScript that should consistently save and then build and run; you can find it in save-and-build.scpt.[2] Note: the script seems to work for me, but I rarely use Xcode to edit my source, so I wouldn't be surprised if it has problems. If you notice one, let me know. If you fix one, I'll update the version in the code distribution.

1. You can find the preference to Always Save Unsaved Files in the Building preferences.
2. You add scripts to Xcode with the Edit User Scripts item in the scripts menu. (That menu is the little scroll shape in the menu bar.) Once you've added a script, you can bind it to a keypress by double-clicking in the second column of the scripts table, pressing the key, and clicking somewhere else.

- If you change—and save—a file within a folder hierarchy but don't change the modification time of the root folder, Xcode won't copy the changed file into the build folder.

Normally, that last problem would be a deal breaker for me. Even if I changed my AppleScript to touch the roots of all folders, I don't want to wait around for Xcode to copy the complete contents of all the folders into the build directory each time. Fortunately, there's a better way to work around the problem. And since my guess is that most of you readers prefer a hierarchical organization for larger projects, I'll use one from now on and show you the workaround.

To use Xcode for the rest of the book, do the following:

1. Set up your rb_main.rb file as described in Section 10.3, *Running in Place*, on the following page.

2. In your moment-to-moment development, use the Debug configuration. (That will activate the code you set up in the previous step.) You are likely using the Release configuration now, so you'll need to change it with Project > Set Active Build Configuration.

3. When you're ready to run the app outside your development folder, make a release by changing the build configuration to Release, cleaning the build (ensuring that everything will be copied into the build folder), and then building the app.

4. Add folders following the procedure given for third-party in Section 9.1, *Preparing a Directory*, on page 111.

5. You can add a file inside a folder in two ways. One is to use File > New File. Don't try it. It will drive you insane.[3] Instead, use the command line to touch the file into existence:

   ```
   $ touch app/Touched.rb
   ```

   After a short time, Xcode will notice and display an icon for the file in its hierarchical view, and you can start editing it. If you're viewing the containing folder in Xcode, you may have to unexpand and reexpand the disclosure triangle before Xcode notices the new file.

3. Think it won't happen to you? OK. But when you get the new file dialog box, make sure you do *not* add it to the project. If you do, the file will be copied into the app twice: once at the top level and once into the folder.

10.3 Running in Place

Suppose your source folder has the subfolders app and third-party. When you make a clean build, those folders will be copied—with the structure unchanged—into the app bundle's Contents/Resources folder. In following builds, Xcode might not notice changes, so it might not update the Contents/Resources folder. However, with a little judicious massaging of load paths by rb_main.rb, the app can be *launched* from the app bundle while *loading* from the source folder—and so it won't matter that the build folder is out-of-date.[4]

That would be a terrible idea for a Release configuration of the app, one that's intended to be moved around and so must be self-contained. But it's fine for the Debug configuration. If you change the rb_main.rb file as follows, it will detect how it's being run and adjust the load paths appropriately:

Download fenestra/editor-agnostic/rb_main.rb

```
require 'osx/cocoa'
require 'path-setting'

if $0 == __FILE__ then
  OSX::NSLog "RubyCocoa version is #{OSX::RUBYCOCOA_VERSION}."
  OSX::NSLog "Using Ruby files in #{RubyCocoaLocations::app_root}."
  RubyCocoaLocations.set_hierarchical_app_load_paths
  RubyCocoaLocations.load_ruby_files
  OSX.NSApplicationMain(0, nil)
end
```

10.4 Building Without Xcode

You may have noticed that Xcode installed a rakefile (named Rakefile) when it created the Fenestra project. The one you will find in fenestra/editor-agnostic is a combination of that one, the version that comes with Rucola,[5] and code I wrote to scratch my own itches. It assumes you've changed your rb_main.rb file as described in Section 10.3, *Running in Place*.

4. I got this clever trick from the Rucola project, http://rucola.rubyforge.org.
5. Rucola helps you organize your project in a way that will be familiar if you've used Rails. Since I didn't want to assume you had, I chose a simpler organization. If you like Rails, you might want to use Rucola's organization instead of mine. Rucola also has its own set of DSL wrappers around Cocoa features like notifications. (I wrote my own because I needed features Rucola's didn't provide.)

During your normal moment-to-moment work, build and run the latest debug version like this:

```
$ rake
(in /Users/marick/.../fenestra/editor-agnostic)
=== BUILDING NATIVE TARGET Fenestra WITH CONFIGURATION Debug ===

Checking Dependencies...
** BUILD SUCCEEDED **
2008-07-12 15:50:42.945 Fenestra[53274:10b] RubyCocoa version is ↩
0.13.1.
2008-07-12 15:50:42.947 Fenestra[53274:10b] Using Ruby files in  ↩
/Users/marick/.../fenestra/editor-agnostic/app.
```

Notice that the app is started up in such a way that debug output flows to the terminal window. Unfortunately, only standard error output goes there. That means you must use either NSLog or $stderr.puts for debugging statements.

When you're ready to distribute a nondebug build, do this:

```
$ rake package
...
=== CLEANING NATIVE TARGET Fenestra WITH CONFIGURATION Release ===
...
=== BUILDING NATIVE TARGET Fenestra WITH CONFIGURATION Release ===
...
  hdiutil create -volname 'Fenestra.2008-07-12' -srcfolder build/R ↩
elease/Fenestra.app pkg/'Fenestra.2008-07-12'.dmg
```

Notice that this puts a timestamped disk image of the releasable application into the pkg folder. If you don't like that, search for :package in Rakefile, and change it. That should be easy to do even if you've never edited a rakefile before.

I tend to use Xcode to do my builds because it usually finishes faster than Rake does. To avoid the annoyance of switching from editor to Xcode, I have my editor[6] set up so that a single keypress pops me into Xcode. From there, it's one more keypress to build.[7]

6. Aquamacs Emacs, http://aquamacs.org/

7. To get back to your editor from Xcode, use Command-Tab. Double-clicking a Ruby file edits in Xcode, even if another editor has claimed it.

10.5 Using Interface Builder with Hierarchical Project Folders

Interface Builder doesn't see Ruby files within subfolders. That means that it doesn't process ib_outlet and ib_target declarations within them. (I think they're still useful as documentation.) You'll have to manually tell IB about outlets and actions via the inspector's Class Identity tab, just as we did with the first GUI way back in on page 45 (and will do again on the following page).

In the past, we've launched Interface Builder by double-clicking a nib file in Xcode. You can also launch it by double-clicking the file in the Finder or by opening it from the command line:

```
$ open English.lproj/MainMenu.nib/
```

As that example shows, the nib file isn't stored at the top level of the project; instead, it's in a localization (.lproj) folder.

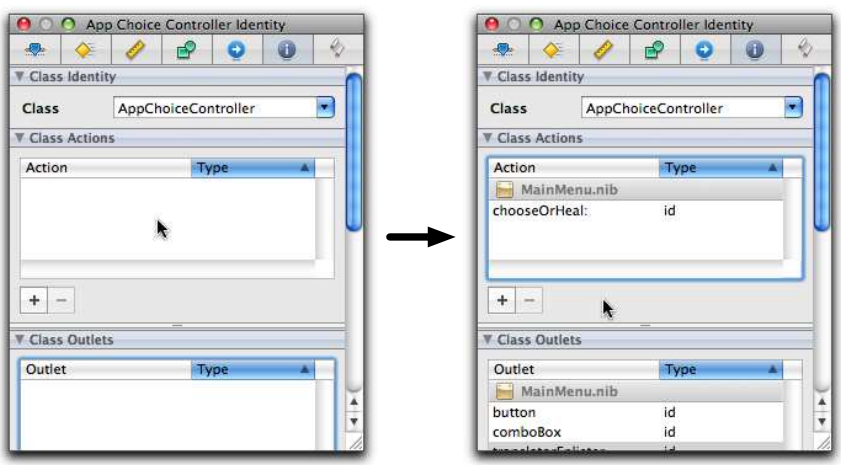
In this chapter, I've moved files around well after Interface Builder had already filled the nib file with information gotten out of them. What happens now that it can't find the Ruby files? That depends on how you start IB:

1. If you start it from outside Xcode, it doesn't try to find the Ruby files. The inspector's Connections tab is unchanged:

   That's fine, because the locations of files doesn't matter to nib loading and object connection. All that matters are the names of classes, methods, and instance variables.

2. If you open Interface Builder from Xcode, it tries to synchronize with Xcode changes, discovers files have (as far as it can tell) been deleted, and displays class connections as suspect:

This does no harm—everything continues to work fine if you build and run. Nevertheless, it makes me a little queasy to see these warnings, so I usually fix them. That's done in the Class Identity tab, which has become empty and needs to be filled:



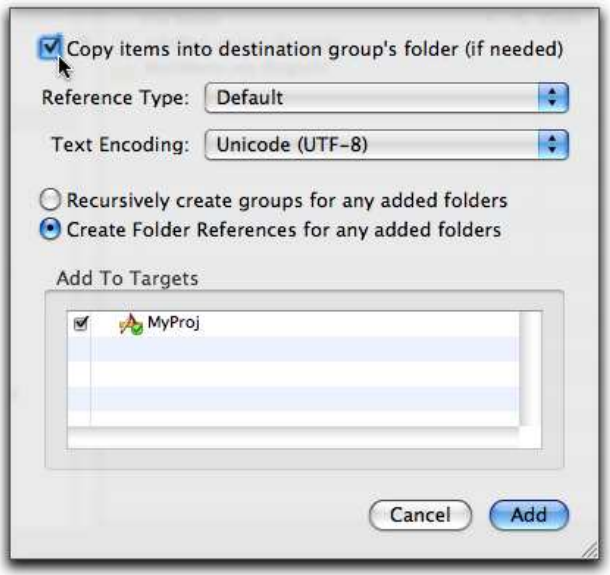
Once that's done, the warnings go away. I've made this fix for you in the editor-agnostic and all later versions of Fenestra.

10.6 Starting a New Project

To get a new RubyCocoa project ready, I do this:

1. Make the project using Xcode as described in Chapter 3, *Working with Interface Builder and Xcode*, on page 37.

2. From the command line, delete Rakefile and rb_main.rb.
3. Use Project > Add to Project to add all the files from the project-files folder. When you add them, be sure you select "Copy files into destination group's folder" and "Create Folder References," as shown here:



4. Change APPNAME in the rakefile to be the name of the project.

The file can now be built with either the rakefile or Xcode. Put Ruby files inside folder app.

10.7 What Now?

Although important, understanding load paths, folder layouts, and build procedures is neither neat, cool, groovy, spiffy, copasetic, nor the cat's pajamas. So, after the drudgery of the last two chapters, you deserve something that's all those things. The next part of the book will cover *Cocoa bindings* and other ways of painlessly synchronizing the user interface to data.

Part IV

Declarative Data Handling

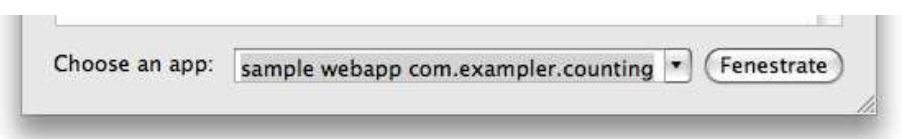
Chapter 11

Persistent User Preferences

Right now, Fenestra has only a short-term memory. The first time you run it, the combo box looks like this:

You might use it to fenestrate a different app by typing something like this:

It will remember that favorite app, but only until you quit. The next time you start Fenestra, the combo box looks like this again:



I think the text field should start out containing my.favorite.app. In this chapter, I'll show you how to save such default values on disk. You could use Ruby's File and IO classes to do that, but Cocoa gives you an NSUserDefaults class that does more of the work for you.

Later in the chapter, we'll change Fenestra so that the user interface pulls data as it needs it. That saves us from having to write controller code that pushes changes into the UI.

11.1 The User Preferences System

Cocoa's *user preferences* or *user defaults* system makes it painless to synchronize in-memory and on-disk versions of user preferences. It's shown in Figure 11.1, on the following page.

❶ An NSUserDefaults object manages preferences. As with NSNotificationCenters, you can have more than one, but you will probably always use the default.

❷ A program uses registerDefaults to tell the app what value to use for a particular keyword if the user has no preference. The argument is an NSDictionary, but it's easier to use Ruby's approximation to keyword arguments (which creates a hash).

❸ A value is fetched from the defaults system with objectForKey. The first time you run Fenestra, it'll use the default preference.

❹ Setting a preference is a simple matter of calling setObject_forKey. The next time objectForKey is called, it will get the new value, even if an app exit and restart happens before then.

Start the version of Fenestra in fenestra/preferred-favorite. Fenestrate some random name of your choosing. (Don't forget to press Return or click the button.) Exit the app and restart—is the name you just typed selected in the combo box?

Preferences like these are stored in the Library/Preferences subfolder in your Home folder. Fenestra's are stored in com.apple.rubycocoa.FenestraApp.plist.[1] It's in a binary format, but double-clicking it will open it in the Property List Editor.

1. You'll see how to change the name in Section 22.7, *Changing the Application's Name*, on page 299.

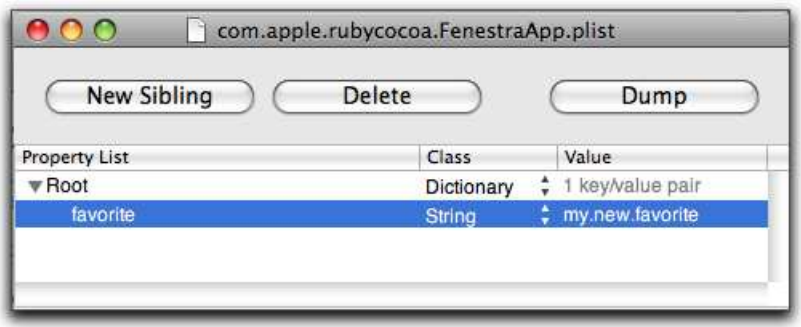
Download fenestra/preferred-favorite/app/TranslatorEnlister.rb

```
class TranslatorEnlister < OSX::NSObject
  DEFAULT_FAVORITE = "sample webapp com.exampler.counting"

  def init
❶   @defaults = NSUserDefaults.standardUserDefaults
❷   @defaults.registerDefaults(:favorite => DEFAULT_FAVORITE)

    k = Struct.new(:display_name, :app_name, :template)
    @translators = [
      k.new(DEFAULT_FAVORITE, "com.exampler.counting", Translators::CountingApp),
    ]

    connect_all_notification_observers

    super_init
  end

❸ def favorite; @defaults.objectForKey(:favorite); end

  on_local_notification AppChosen do | notification |
    display_choice = notification[:app_name].to_ruby
❹   @defaults.setObject_forKey(display_choice, :favorite)
    translator = @translators.find { | t |  t.display_name == display_choice }
    if translator
      translator.template.alloc.initForApp(translator.app_name).listen
    else
      Translators::ToString.alloc.initForApp(display_choice).listen
    end
  end
end
```

fenestra/preferred-favorite/app/TranslatorEnlister.rb

Figure 11.1: Saving the favorite

There, after a little bit of tree expansion, you'll see this:



Try This Yourself

1. Run the app, change its favorite, and force-quit it. Are changed preferences written out immediately or on app exit? Perhaps preferences are written out at periodic intervals: what happens if you let the app run for five or more minutes and then force-quit it?

2. You can use the user defaults system from irb, but you must explicitly update the on-disk version with the synchronize method:

```
$ irb
irb(main):001:0> d = OSX::NSUserDefaults.standardUserDefaults
=> #<OSX::NSUserDefaults:0x2c1e1a class='NSUserDefaults' id=0x279e00>
irb(main):002:0>  d.setObject_forKey('dawn', :favorite)
=> nil
irb(main):003:0> d.synchronize
=> true
irb(main):004:0> exit
$ irb
irb(main):001:0> d = OSX::NSUserDefaults.standardUserDefaults
=> #<OSX::NSUserDefaults:0x2c1e1a class='NSUserDefaults' id=0x279e00>
irb(main):002:0> d.objectForKey(:favorite)
=> #<NSCFString "dawn">
```

   Use irb to convince yourself that preferences do not have to be strings; they can also be integers, floating-point numbers, Time objects, true, false,[2] hashes and arrays (as long as the latter two contain only objects in this list), and raw data (encapsulated in NSData objects).

   What happens if you try to store something not in the list, like a Range? Or an NSNotification?

2. But see Section 11.2, *Booleans Again*, on page 136.

3. NSUserDefaults objects respond to type-specific methods like stringForKey and integerForKey. Unlike objectForKey, these do type conversions if needed. To see that in action, store a string *"33"* and retrieve it with integerForKey. Also do the reverse: store the integer 33 and retrieve it with stringForKey.

4. You may not know that preferences can be manipulated from the command line with the defaults command. Try reading and then changing the favorite like this:

```
$ defaults read com.apple.rubycocoa.FenestraApp
{
    favorite = my.new.favorite;
}
$ defaults write com.apple.rubycocoa.FenestraApp favorite commander
$
```

   Is the change reflected in Fenestra when you restart it?

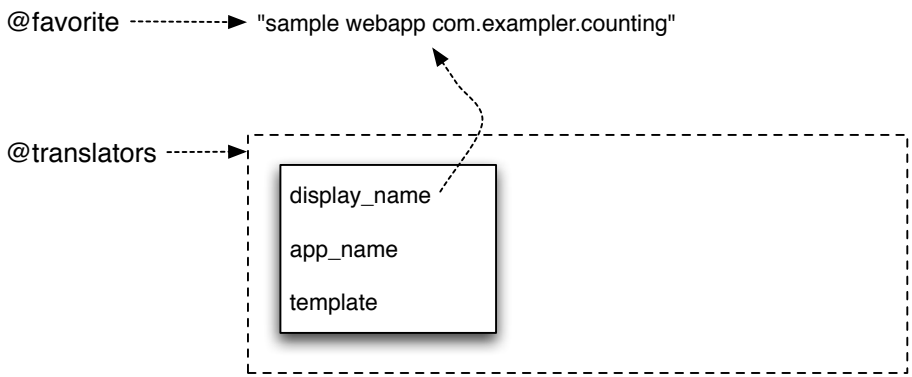
11.2 Storing Custom Objects as Preferences

Run Fenestra, and add a new preferred app. Click the combo box's down arrow to see the list of known apps. Notice that it hasn't been updated with the new preference:

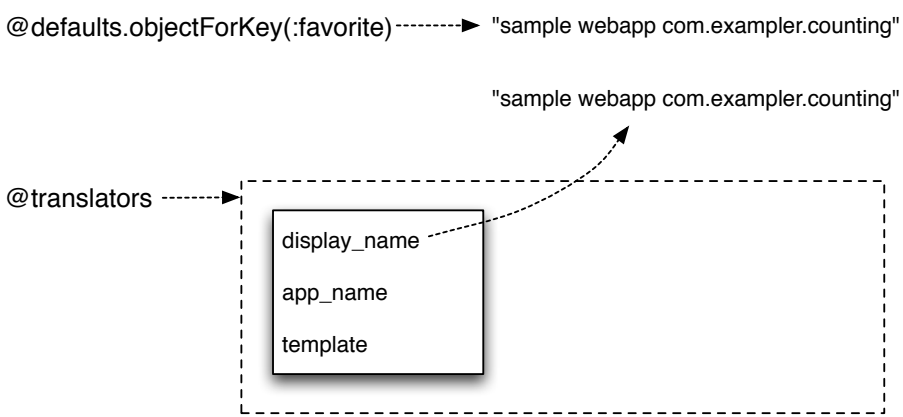
It's not surprising. None of the code changes (from Figure 11.1, on page 129) touched the combo box. We'll fix that next.

Data Structure

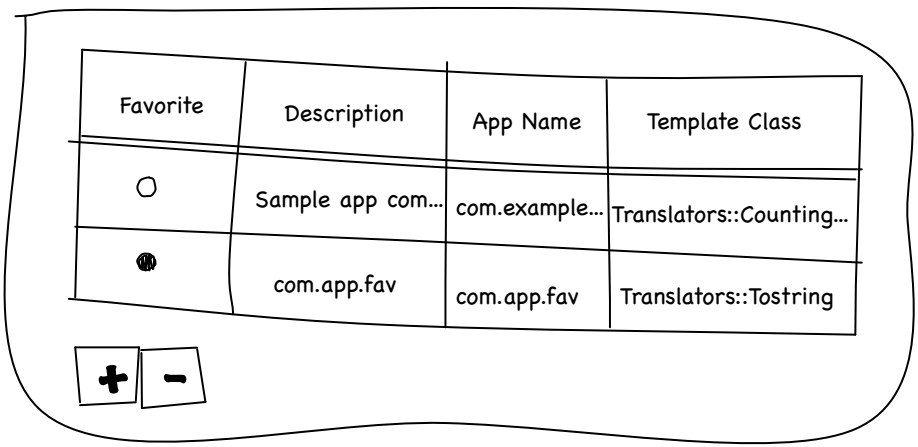
Before starting, I want to fiddle with what I have. In the pre-preferences version of Fenestra, a TranslatorEnlister had two instance variables that looked like this:



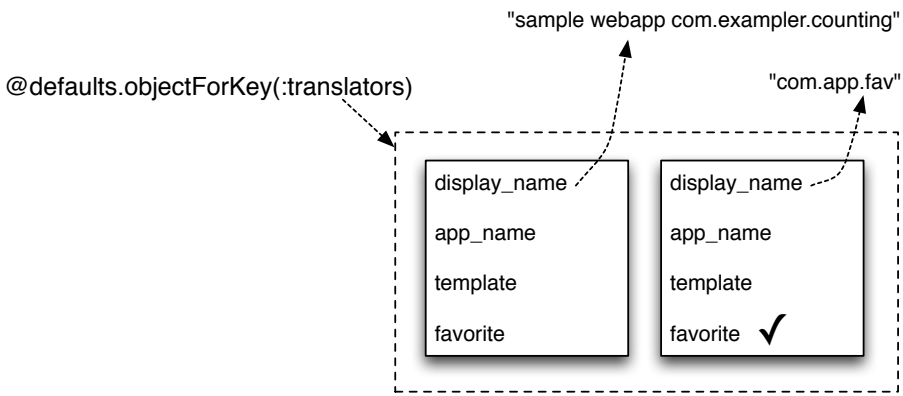
The display name of the favorite is retrieved via two routes: via variable @favorite for the text field aspect of the combo box and via @translators for the list aspect. That's been making me a touch queasy for a while. The current version makes me feel worse. The string fetched by objectForKey is a different string than the one in the @translators list, even though they have the same value:



Seeing the same information in two places makes creepy "something bad is going to jump out at you" music start playing in my head. We could store the array index of the favorite as the :favorite default, but I'm going to use a different solution. This part of the book is building toward a Fenestra preference pane that I imagine looking something like this, where the favorite is chosen with a radio button:



So, it seems natural to echo that tabular structure in the code.[3] Here's the structure I propose:



3. As I wrote that sentence, my copy of Eric Evans's *Domain-Driven Design* jumped off the bookshelf and stalked angrily out the door. If books could talk, it would have said that an app's internal structure shouldn't be based on the accidents of user interface design or on-disk/database storage layout. To see why, consider that the new data structure makes it possible for a user to have more than one favorite. Whereas before changing the favorite was a simple assignment that automatically erased the old favorite, now the code will have to specifically erase it. More code...more chance of error. (To which I might have replied—if I talked back to books—that my daughter seems to have no problem having five best friends. Maybe I've stumbled on a useful concept in the domain. . . .)

And here's the class definition we'll be working with:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
class TranslatorPreference < OSX::NSObject
  Properties = [:display_name, :app_name, :class_name, :favorite, :source]
  attr_accessor *Properties
end
```

I've declared the attributes in a rather odd way because I want my later methods to look like this:

```
Properties.each do | prop |
   do_something_with(prop)
end
```

Not only will that save me typing, but I won't have to remember to tweak each of the methods whenever I add a new attribute.

The attributes are almost the ones you've seen before, except for two changes:

class_name
: This replaces template. In addition to being a dumb name, template pointed at a Class object (like Translators::ToString). Later in the book, when Fenestra lets the user type a new class name into a text field, it just became massively more convenient to keep around the name of the class instead of the class itself. So, I reversed time, came back to this moment, and created class_name to hold a string.

source
: This holds the location of a user-defined translator. As with class_name, it's more convenient to add it now than later.

I chose the name Properties because that's what Cocoa documentation calls what we Rubyists call *attributes*.

Archiving

Now we just have to find a way to stuff TranslatorPreference objects into the user preferences system. The problem is that NSUserDefaults works with only a small set of classes, and TranslatorPreference isn't one of them. In the Cocoa world, a conventional workaround is to convert the objects into NSData before storing them as preferences. That's what we will do.

What other class libraries call *marshaling*, *pickling*, or *serializing*, Cocoa calls *archiving* or *coding*.[4] Like most such schemes, archiving requires cooperation from the classes being archived. In particular, they must implement encodeWithCoder. Here's TranslatorPreference's implementation:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
def encodeWithCoder(coder)
  Properties.each do | prop |
    value = self.send(prop)                    ❶
    coder.encodeObject_forKey(value, prop)     ❷
  end
end
```

The tricky bit is at ❶. By sending the name of the property to the object itself, I'm calling the reader. That is, for the first Property, the marked line is equivalent to this:

```
value = self.display_name
```

At ❷, I just encode each field with a built-in NSCoder method. It knows how to encode certain objects—strings and numbers and booleans, for example—and recursively calls encodeWithCoder on any it doesn't know about.

Unarchiving is done by first creating an instance with alloc and then calling initWithCoder instead of the usual init method. Here's TranslatorPreference's implementation:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
def initWithCoder(coder)
  hash = {}
  Properties.each do | prop |
    hash[prop] = coder.decodeObjectForKey(prop)
  end
  initWithHash(hash)
end
```

4. Why two terms? There are several classes that implement the idea. The one preferred by Apple's *Archives and Serializations Programming Guide for Cocoa* [App08c] is NSKeyedArchiver. It's a subclass of NSCoder. *Archiver* is probably the better name, but *coder* is the older one. So—as you'll see in a moment—it has been immoralized in various method names.

It collects up a hash, something like this:

```
{
  :display_name => "sample webapp com.exampler.counting",
  :app_name => 'com.exampler.counting',
  ...
}
```

That hash is used in initWithHash:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
def initWithHash(hash)
  Properties.each do | prop |
    writer = prop.to_s + '='
    self.send(writer, hash[prop])
  end
  self
end
```

For the first Property, the body of the each is equivalent to this:

```
self.display_name = "sample webapp com.exampler.counting"
```

Booleans Again

As it turns out, I had to override the default writer for favorite:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
def favorite=(value)
  @favorite = value
  @favorite = false if @favorite == 0
end
```

Why is that needed? The idea of "falsity" is represented in Objective-C by a raw machine word containing zero, not by an object. When the Ruby object false moves into the Objective-C universe, it's converted into an NSCFBoolean object. You can see that like this:

```
irb(main):001:0> v = false.to_ns
=> #<NSCFBoolean false>
```

Although this behaves as a boolean, it's actually a number:

```
irb(main):002:0> v.class.superclass
=> OSX::NSNumber
irb(main):003:0> v.intValue
=> 0
```

An NSCFBoolean is a legitimate object, not a machine integer, but it's converted into a machine integer when that's needed. Evidently it's needed somewhere during the process of being archived, put into user preferences, recovered from user preferences, and unarchived, because the "booleanness" of favorite gets lost. It's turned into one of the Ruby

integers 0 or 1. Since 0 is "true" to Ruby, I have to manually change it to false. The lesson: in your RubyCocoa career, expect to be puzzled by odd behavior only to discover it's because of a round-trip conversion of false to 0.

Manual Transformations

Back in the day when class_name was template, I used it as an example of saving an object that isn't built-in, as strings and numbers are, but that didn't implement the methods archivers require. It's simple enough: you just have to convert the object into some archivable form and archive that. For example, the following method handles a template by storing its name (not including the Translators:: at the beginning):

```
def encodeWithCoder(coder)
  ...
  name = @template.name.split('::').last
  coder.encodeObject_forKey(@template.name, :template)
  ...
end
```

It can be unarchived by looking up the name in the Translators module:

```
def initWithCoder(coder)
  ...
  name = coder.decodeObjectForKey(:template)
  @template = Translators.const_get(name)
end
```

You'll get to practice with a different type of unarchivable object in Section 11.5, *Try This Yourself: A Sticky Window*, on page 145.

Printing NSObjects

On occasion, the Cocoa runtime will print information about an object (such as when there's an error). Unlike the Ruby runtime, it doesn't get a string to print by calling to_s. Instead, it calls description. By default, an NSObject's to_s method calls description, so description is the method to define if you want your objects printed prettily. Here's the definition of description for TranslatorPreference:

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
def description
  "<#{self.class}: #{display_name}/#{app_name}/#{class_name}" +
    " favorite: #{favorite} from '#{source}'."
end
alias inspect to_s
```

You can see the complete definition of TranslatorPreference in Figure 11.2, on the next page.

Download fenestra/preferred-list/app/preferences/TranslatorPreference.rb

```
class TranslatorPreference < OSX::NSObject
  Properties = [:display_name, :app_name, :class_name, :favorite, :source]
  attr_accessor *Properties

  # Override writers that need special handling
  def favorite=(value)
    @favorite = value
    @favorite = false if @favorite == 0
  end

  def initWithHash(hash)
    Properties.each do | prop |
      writer = prop.to_s + '='
      self.send(writer, hash[prop])
    end
    self
  end

  def encodeWithCoder(coder)
    Properties.each do | prop |
      value = self.send(prop)
      coder.encodeObject_forKey(value, prop)
    end
  end

  def initWithCoder(coder)
    hash = {}
    Properties.each do | prop |
      hash[prop] = coder.decodeObjectForKey(prop)
    end
    initWithHash(hash)
  end

  def description
    "<#{self.class}: #{display_name}/#{app_name}/#{class_name}" +
      " favorite: #{favorite} from '#{source}'."
  end
  alias inspect to_s
end
```

fenestra/preferred-list/app/preferences/TranslatorPreference.rb

Figure 11.2: A class that can be stored as a preference

11.3 Using Archived Objects

There's something that has been bothering me about TranslatorEnlister for a while. Although it sits below the "presentation layer," it knows facts about presentation, specifically, strings to display in the UI. Earlier in this chapter, it grew a little knowledge about user preferences. If I follow my current path and have it pull in and decode TranslatorPreferences, it's going to drift even further from the "do one thing and do it well" ideal for classes.

So, what I'll do is create a new class, Preferences. Its job is to answer questions other classes have about user preferences. Here are some questions AppChoiceController will have for it:

- What's the name of the favorite translator? I need to display it in the text box.
- What's the list of names I should put in the drop-down list part of the combo box?
- Can you point me at whichever name is the favorite? I need to highlight that list entry.

Moving Responsibilities Around

I'll start designing Preferences by changing AppChoiceController to use it and seeing what method names fall out. Here's my first cut:

Download fenestra/preferred-list/app/controllers/AppChoiceController.rb

```
class AppChoiceController < Controller
  # ...
❶ ib_outlet :preferences

  def awakeFromNib
❷   @comboBox.stringValue = @preferences.display_name_of_favorite_translator
    fill_combo_box
    super
  end

  def fill_combo_box
    @comboBox.removeAllItems
❸   @preferences.translator_display_names.each do | t |
      @comboBox.addItemWithObjectValue(t)
    end
  end

  # ...
end
```

❶ As was the case with the TranslatorEnlister, the Preferences object is connected as an outlet. That means it has to be added to Interface Builder's doc window (as described in Section 3.1, *Connecting the Interface to Code*, on page 44).

❷ The AppChoiceController asks Preferences for the text to put in the text field. (It used to ask the TranslatorEnlister.) I changed the name from favorite to display_name_of_favorite_translator because, first, the original name was ambiguous[5] and, second, because the Preferences may eventually answer many questions from many classes about what the user prefers, so it's best to make method names painfully explicit.

❸ Similarly, I ask the Preferences for a list of all the display names. You and I know that those names are plucked from TranslatorPreferences, but there's no reason for the AppChoiceController to know anything about that class.

I've put this code in its own method because it's not just called from awakeFromNib. The combo box needs to be filled again after a new favorite is chosen.

If you browse TranslatorEnlister now, you'll see that all that's left for it to do is this:

Download fenestra/preferred-favorite/app/TranslatorEnlister.rb

```
on_local_notification AppChosen do | notification |
  display_choice = notification[:app_name].to_ruby
  @defaults.setObject_forKey(display_choice, :favorite)
  translator = @translators.find { | t |  t.display_name == display_choice }
  if translator
    translator.template.alloc.initForApp(translator.app_name).listen
  else
    Translators::ToString.alloc.initForApp(display_choice).listen
  end
end
```

It fiddles with the favorite display name (which is no longer its business), it finds the appropriate translator (which probably should not be its business, since that involves the user's preferences), and it launches the translator and forgets about it. If we move the preferences code into the Preferences class, that leaves practically nothing.

5. Did I want the whole TranslatorPreference? Its app_name? Its display_name? What?

Let's just destroy TranslatorEnlister—I never really liked it anyway—and move translator launching into AppChoiceController:

Download fenestra/preferred-list/app/controllers/AppChoiceController.rb

```
ib_action :chooseOrHeal do | sender |
  if @button.state == NSOnState
    @last_choice = @comboBox.stringValue.to_ruby
❶   translator = @preferences.translator_for_display_name(@last_choice)
❷   translator.listen
❸   fill_combo_box
    post(AppChosen, :app_name => @last_choice)
  else
    post(TimeToForgetApp, :app_name => @last_choice)
  end
end
```

AppChoiceController obeys user preferences at ❶ and ❷. I still don't want it to know anything about the details of preferences, so I made the query method return an already-initialized Translator that just has to be told what to do.

Notice that I've pushed the knowledge of what makes a Translator the current favorite into Preferences. Since that class is the authority over what the user prefers, it should also be the authority over what makes that preference change.

After changing the favorite, I call fill_combo_box at ❸ to update the combo box with a (potentially) new list of translators. . . but wait! I just isolated knowledge about how that list changes by putting it inside Preferences. But, moments later, I've written code that AppChoiceController executes because the list might change. It's acting on knowledge it's not supposed to have.

This design *works*, but it's sloppy in a way that makes bugs more likely as Fenestra changes. We'll fix that in Section 11.4, *Views Can Pull Data*, on page 143, but first we'll finish implementing the design we have.

Implementing the Adapter

In jargon,[6] Preferences is an *adapter*, a class that improves on an interface that it hides. Our adapter's first job is to register the default preferences. In what method should that be done?

6. See *Design Patterns in Ruby* [Ols07].

In my first implementation, I foolishly put it in awakeFromNib. The problem is that objects may be awakened in any order. At the moment any object's awakeFromNib runs, its outlets are connected to the appropriate objects, but there's no guarantee those other objects have run *their* awakeFromNib methods. So, this sequence of events was possible:

1. The Preferences and AppChoiceController are created. (The order doesn't matter.)
2. AppChoiceController's awakeFromNib is called. It uses its @preferences outlet to ask how to set up its combo box.
3. But Preferences hasn't run its awakeFromNib, so it hasn't registered the default preferences.
4. AppChoiceController gets back nil values.

I was lucky: events happened in that order when I first ran the changed Fenestra. If the two class's awakeFromNib methods had been run in the other order, I might not have found out about the bug until a reader like you found it.[7]

After that discovery, I put the code in an init method:

Download fenestra/preferred-list/app/preferences/Preferences.rb

```
def init
  only = TranslatorPreference.alloc.initWithHash(
              :display_name => "sample webapp com.exampler.counting",
              :app_name => "com.exampler.counting",
              :class_name => 'CountingApp',
              :favorite => true,
              :source => nil)
  @defaults = NSUserDefaults.standardUserDefaults
  @defaults.registerDefaults(:translators => collect_archived([only]))
  super_init
end
```

This uses collect_archived to produce an array of archived objects. It happens to have only one, our old friend com.exampler.counting.[8]

collect_archived uses a new Cocoa class we haven't seen before, NSKeyedArchiver, to archive each TranslatorPreference (using that class's encodeWithCoder method).

7. I've made this same mistake more times than I want to admit. I hope the danger of awakeFromNib soaks into your brain better than it has into mine.
8. In previous versions of Fenestra, the last element of the drop-down list was "for other apps: use.dot.format.name." I've gotten tired of that, so I'm dropping it.

Download fenestra/preferred-list/app/preferences/Preferences.rb

```
def collect_archived(objects)
  objects.collect do | o |
    NSKeyedArchiver.archivedDataWithRootObject(o)
  end
end
```

After the object is initialized, each of Preferences externally visible methods starts by reading preferences with this code:

Download fenestra/preferred-list/app/preferences/Preferences.rb

```
archived = @defaults.objectForKey(:translators)
@translator_preferences = archived.collect do | nsdatum |
  NSKeyedUnarchiver.unarchiveObjectWithData(nsdatum)
end
```

If the method could have changed the preferences, it finishes with this code:

Download fenestra/preferred-list/app/preferences/Preferences.rb

```
@defaults.setObject_forKey(collect_archived(translator_preferences),
                                            :translators)
```

The methods that contain that code are mildly tricky, but they will teach you nothing new about Cocoa or RubyCocoa. If you care to read them, I added some comments explaining the cute parts.

Try This Yourself

1. Run the new version of Fenestra. Type a new name in the dialog box, and click the Fenestrate button. Look at the combo box's drop-down list. Does it contain the new name?

   Now quit Fenestra, and restart it. What's the value in the text field? Is your new app name in the drop-down list?
2. Use the Property List Editor to see what NSData looks like inside a preferences file. Not so useful for debugging, eh?

11.4 Views Can Pull Data

Some Cocoa views can be told to pull the data they display from a *data source* whenever they paint themselves on the screen. That way, their controllers don't have to make sure to update them at all the right moments. In our case, it means that AppChoiceController doesn't have to know when user preferences might change.

Lines ❶ and ❷ show how AppChoiceController can tell its combo box to use the data it (self) will provide:

Download fenestra/preferred-pull/app/controllers/AppChoiceController.rb

```
def awakeFromNib
❶   @comboBox.usesDataSource = true
❷   @comboBox.dataSource = self
❸   @comboBox.stringValue = @preferences.display_name_of_favorite_translator
    super
end
```

Notice also that the combo box's value-as-a-text-field still needs to be set. Were the line at ❸ removed, the text box would be initialized with the drop-down list's first entry, not with the favorite.

Being a Data Source

A combo box uses a data source by iterating over its contents. To do that, it has to know how big the list could be:

Download fenestra/preferred-pull/app/controllers/AppChoiceController.rb

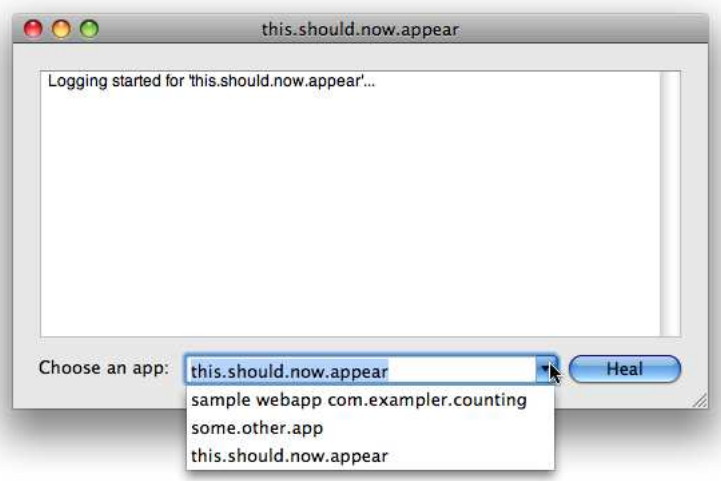
```
def numberOfItemsInComboBox(ignored)
  @preferences.translator_display_names.size
end
```

And it must be able to ask for each item by its index:

Download fenestra/preferred-pull/app/controllers/AppChoiceController.rb

```
def comboBox_objectValueForItemAtIndex(ignored, index)
  @preferences.translator_display_names[index]
end
```

These two methods are enough to fill a combo box:



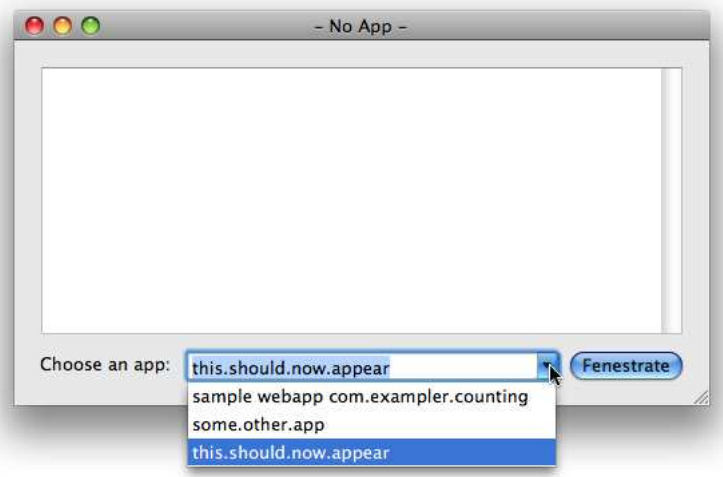
The favorite app appears in both the text field and the drop-down list, but it's not highlighted in the latter. An optional method adds that bit of polish:

Download fenestra/preferred-pull/app/controllers/AppChoiceController.rb

```
def comboBox_indexOfItemWithStringValue(ignored, string)
  @preferences.translator_display_names.index(string) || NSNotFound
end
```

When the drop-down list is displayed, the combo box asks this method which entry should be highlighted. If none matches, the method returns NSNotFound.[9]

List-element highlighting now works:



If you try it, you'll see it works well with autocompletion, too. The combo box asks for the list index after each character typed.

11.5 Try This Yourself: A Sticky Window

Here's a coding task that involves user preferences but also gives you a chance to refresh your memory of previous topics.

You've probably noticed that most Mac apps have "sticky" windows. If you move them around, they'll appear in their new positions the next

9. Ruby methods that return indices normally return nil to mean "not found." NSNotFound is not equal to nil, so be careful.

time you start the app. That works when apps store the window's *frame* (a rectangle that surrounds it) in user preferences.

Fenestra's single window doesn't stick. There's an easy fix for that; you'll learn it in Chapter 22, *Fit and Finish*, on page 287. For now, though, do it by manually saving and loading the data in the user preferences.

Hints

- The window's frame is retrieved by sending it the frame message. It returns an NSRect, which is not one of the data types that can be stored as a preference. It also can't be archived into NSData in the way that TranslatorPreference was. That's because it's not a real object. Instead, it's a struct, like NSRange.[10] You'll need to convert the frame into some format that can be stored, such as an array of floats.[11]
- When restoring the frame, use setFrame:display:. The first argument is an NSRect, and the second is a boolean that determines whether the window should redraw all its subviews. (Try it both ways and see whether it makes a difference.)
- You can choose to store a changed window frame either before exiting or at the moment it changes. A class can track such changes by observing NSWindowDidMoveNotification notifications.

My Solution

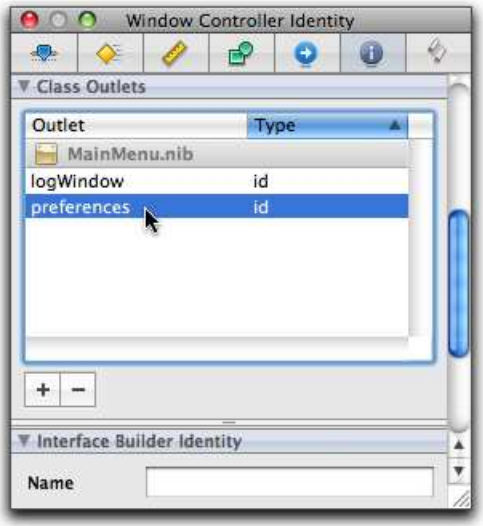
I decided to use Preferences to store the frame and WindowController to initialize the window. That meant I needed to make a new outlet from WindowController to Preferences:

Download fenestra/preferred-frame/app/controllers/WindowController.rb

```
class WindowController < Controller
  ib_outlet :logWindow, :preferences
```

Because I've been using a hierarchical project folder, Interface Builder can't notice the outlet declaration, so I had to tell it manually.

10. Structs were described on page 54.
11. You can convert an NSRect into an array of floats with to_a. If you don't want to mess with conversions, you'll find that you can ask for the frame as a string. You use stringWithSavedFrame. You can restore the frame with the same string using setFrameFromString:.



Then it was a simple matter of connecting the two classes.

I made one change to WindowController. It sets its logWindow's frame when it awakens:

Download fenestra/preferred-frame/app/controllers/WindowController.rb

```
def awakeFromNib
  @logWindow.setFrame_display(@preferences.sole_window_frame, true)
end
```

As far as I can tell, the second argument is irrelevant to this case.

The window position default preference is set in the Preferences object's init method so that it's available when AppChoiceController wants it. The work is done at lines ❶ and ❷:

Download fenestra/preferred-frame/app/preferences/Preferences.rb

```
def init
  only = TranslatorPreference.alloc.initWithHash(
              :display_name => "sample webapp com.exampler.counting",
              :app_name => "com.exampler.counting",
              :class_name => 'CountingApp',
              :favorite => true,
              :source => nil)
❶   window_frame = [196.0, 237.0, 520.0, 295.0]
  @defaults = NSUserDefaults.standardUserDefaults
  @defaults.registerDefaults(:translators => collect_archived([only]),
❷                            :frame => window_frame
                            )
  # ...
end
```

There's a problem with this code: any window's starting position is stored in the nib file. (That's where I got the values for ❶—they're from the Size [third] tab in the inspector.) That duplication means that someone wanting to change the default starting position will almost certainly change it in IB and then be confused when nothing happens. But hard-coding these values is less work than writing code to deal with the nil value returned when an NSUserDefaults has no matching key, and we're going to discard this solution anyway in Chapter 22, *Fit and Finish*, on page 287.

Because I decided earlier that Preferences should be the sole authority about when a user's translator preferences change, I also made it the authority over what happens when she moves a window. The Preferences object listens for window movement notifications, converts them to an array of floats, and stores them:

Download fenestra/preferred-frame/app/preferences/Preferences.rb

```
on_local_notification NSWindowDidMoveNotification do | notification |
  frame = notification.object.frame
  array = [ frame.origin.x, frame.origin.y,
            frame.size.width, frame.size.height]
  @defaults.setObject_forKey(array, :frame)
end
```

Since I'm using my little DSL for notifications, I have to do work in init:

Download fenestra/preferred-frame/app/preferences/Preferences.rb

```
def init
  # ...
  connect_all_notification_observers
  super_init
end
```

Finally, in sole_window_frame, I reassemble the array into an NSRect:

Download fenestra/preferred-frame/app/preferences/Preferences.rb

```
def sole_window_frame
  x, y, width, height = @defaults.objectForKey(:frame)
  NSRect.new(NSPoint.new(x, y), NSSize.new(width, height))
end
```

Notice that this solution will break if I ever add another window. Since we'll be discarding it soon enough, I won't worry about that.

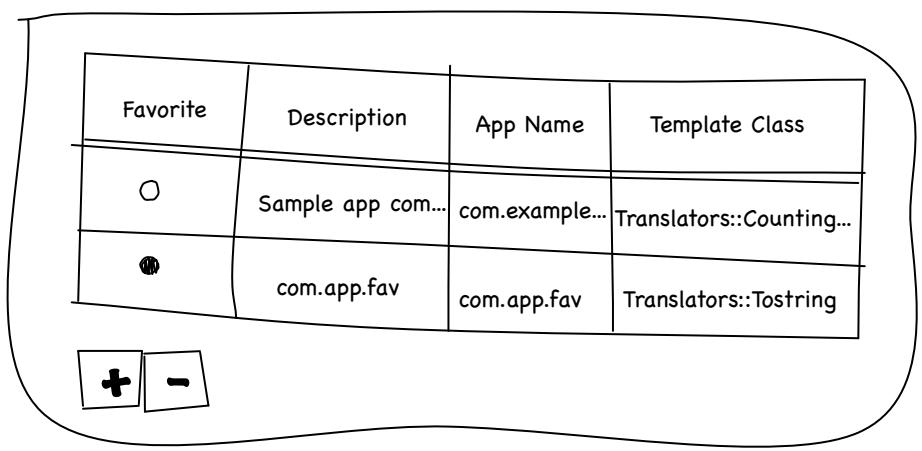
11.6 What Now?

In this chapter we have one-way synchronization: a UI control that automagically pulls data from a data source. The next chapter sets the stage for two-way synchronization: making it so that a change to either side of the relationship propagates to the other. In many cases, that can be done without writing any code at all.

Chapter 12

Creating a Preference Panel in a New Nib

In this chapter, we'll implement a first version of this preference panel:

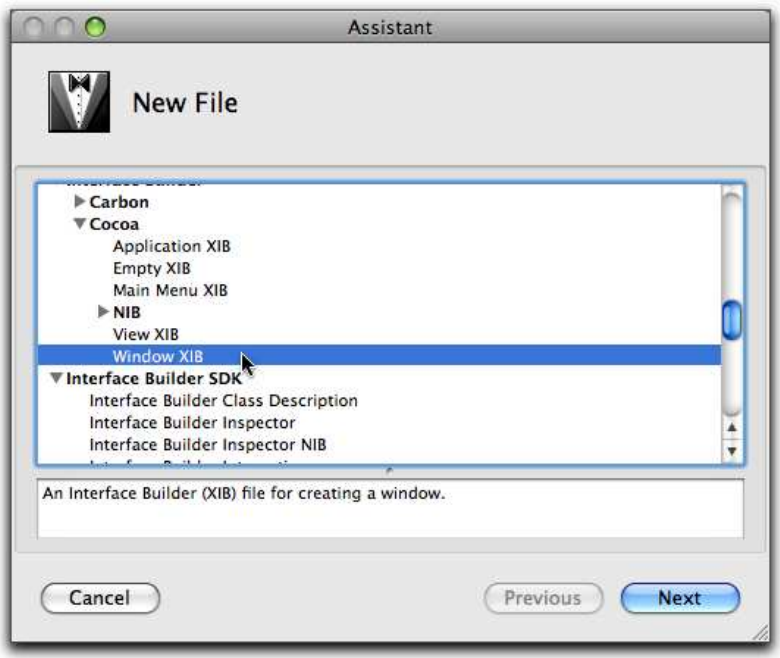


In the next chapter, you'll use it to learn about Cocoa's support for synchronizing UI controls and program data. This chapter is only about creating it.

We're going to put the panel in a new nib file. If we put it in the main nib, all the objects that make up the preference panel would be loaded up and connected each time Fenestra ran, possibly noticeably slowing down app startup. After that, they'd hang around—invisibly—waiting for the user to want to edit preferences. Since that's something users rarely do, the slowdown would be a waste.

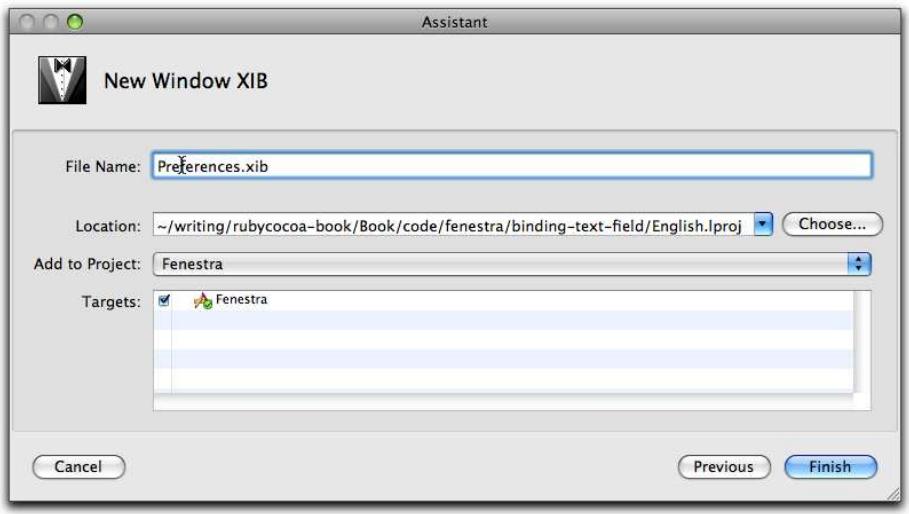
12.1 Creating a Nib

Start from fenestra/preferred-pull or your own equivalent. Create the nib file with Xcode's File > New File. Navigate through the tree control until you're creating a Window XIB. (You'll have to do a little tree expansion.) That looks like this:

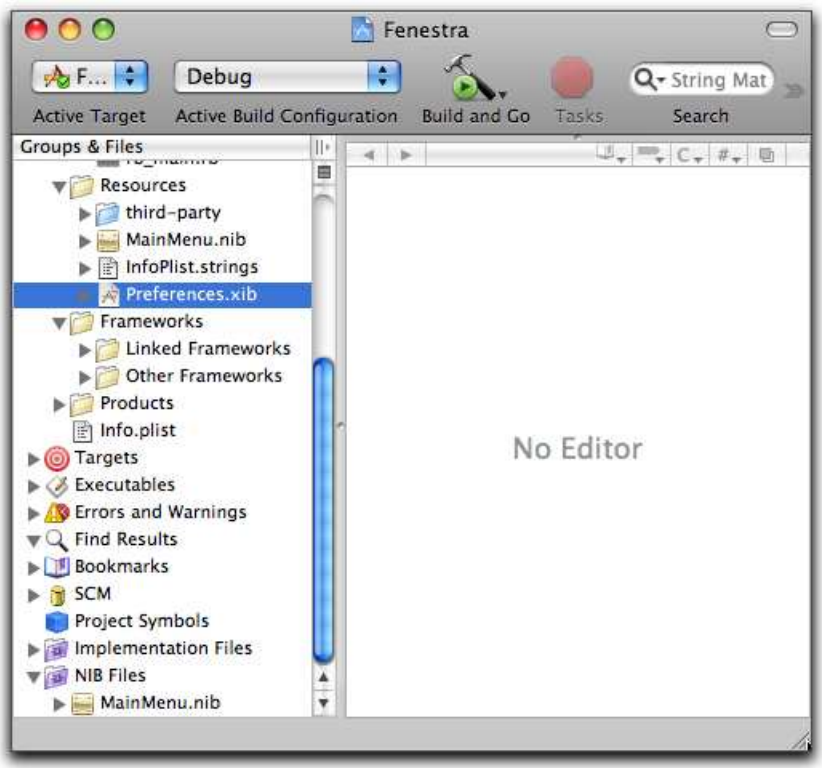


A xib file is just like a nib file, except it is purely text, so it plays better with version-control systems like Subversion or Git.

Name the new file Preferences.xib, and put it in the project's English.lproj directory:



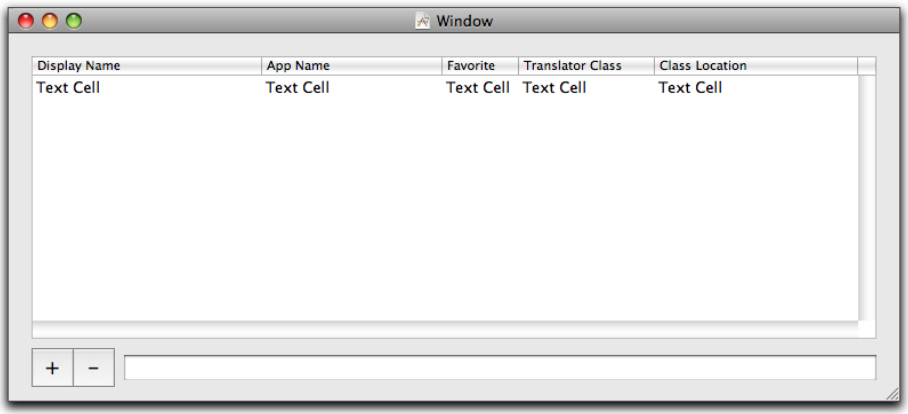
When you create the xib file, Xcode will show it somewhere in the main browser window, probably not where you want it. Drag it under Resources:



It may also appear under the NIB Files smart group (near the bottom of the window) alongside MainMenu.nib. If it doesn't and you want it to, Control-click the smart group, pick menu item Get Info, and change the smart group's pattern from *.nib to *.*ib.

12.2 Drawing the Panel

In IB, change the window to look like this:



That will involve the following:

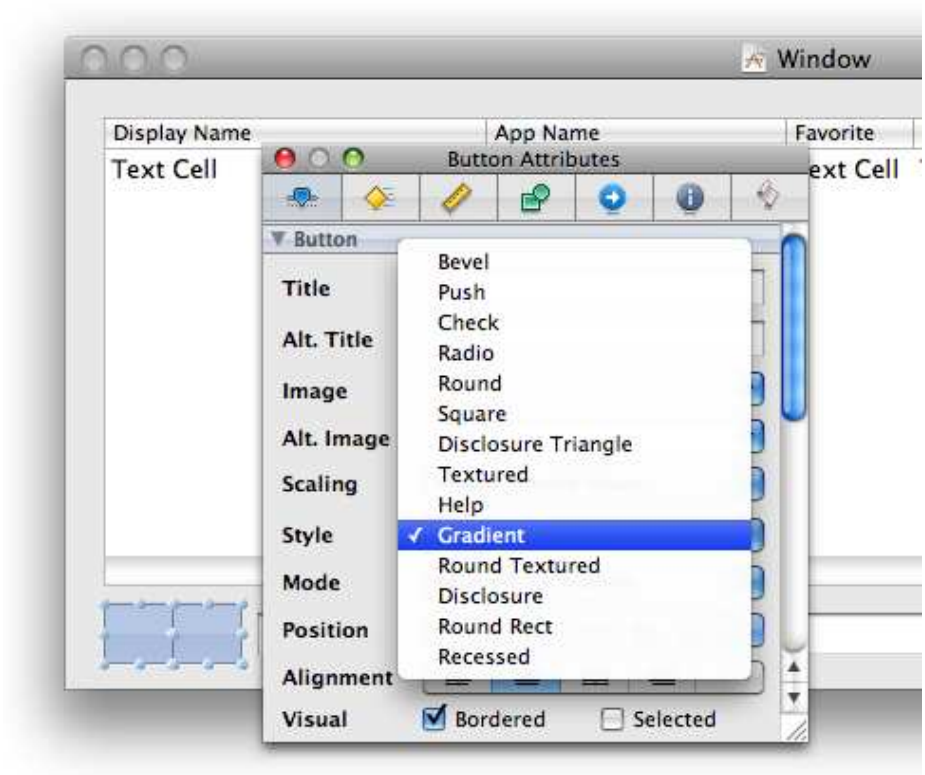
1. Changing the window's class to NSPanel (in the inspector's Window Identity tab). An NSPanel adds a little behavior to an NSWindow, none of which we'll actually use.
2. Dragging an NSTable from the object library.
3. Dragging two NSButton templates from the library. The template you want is labeled Square Button. Leave the buttons blank for the moment.
4. Dragging in an NSTextField.
5. Giving the table five columns (with its Attributes inspector). The current version of TranslatorPreferences has only four attributes. The extra column will be used for a fifth, which will hold the location of the Ruby source for a custom translator.
6. Naming each column. The easy way to do that is to double-click the header cell to type in the name. You can also select the whole column by clicking slowly two or more times below the "Text Cell" text. That will highlight the whole column and open the inspector

on it. Then you can use the Attributes tab to set the title. (Take care when selecting a column this way—it's easy to get a cell or the whole table instead. Then hilarity ensues when the attribute you expect is missing or—worse—you set an attribute on the wrong UI element.)

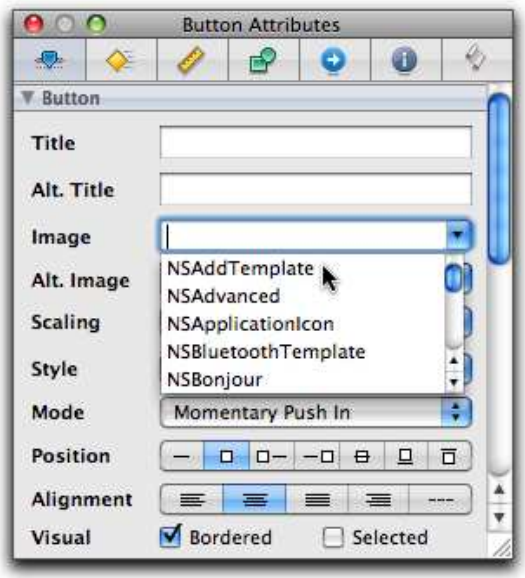
7. Using the panel's Size inspector to move it somewhere not completely on top of the main window. (Leave a little overlap, though, because that will help you see which window is on top.)

Images in Buttons

When buttons affect a view (by, for example, adding or removing table rows), the Apple Human Interface Guidelines recommend gradient buttons instead of square buttons. I had you drag out a square button because the gradient button template is harder to work with. You can change the buttons to gradient buttons by selecting both of them and then making the change in the Attributes inspector:



Gradient buttons are intended to contain images rather than text. A large number of images are predefined, including the ubiquitous + and - signs. You select these images from the Attribute inspector's Images drop-down. The + image is named NSAddTemplate:



If the image in the button is off-center in any direction, check the Position row in the Attributes inspector:

It controls the relative position of text (represented by the line) and image (represented by the square). The second, image-only entry should be selected.

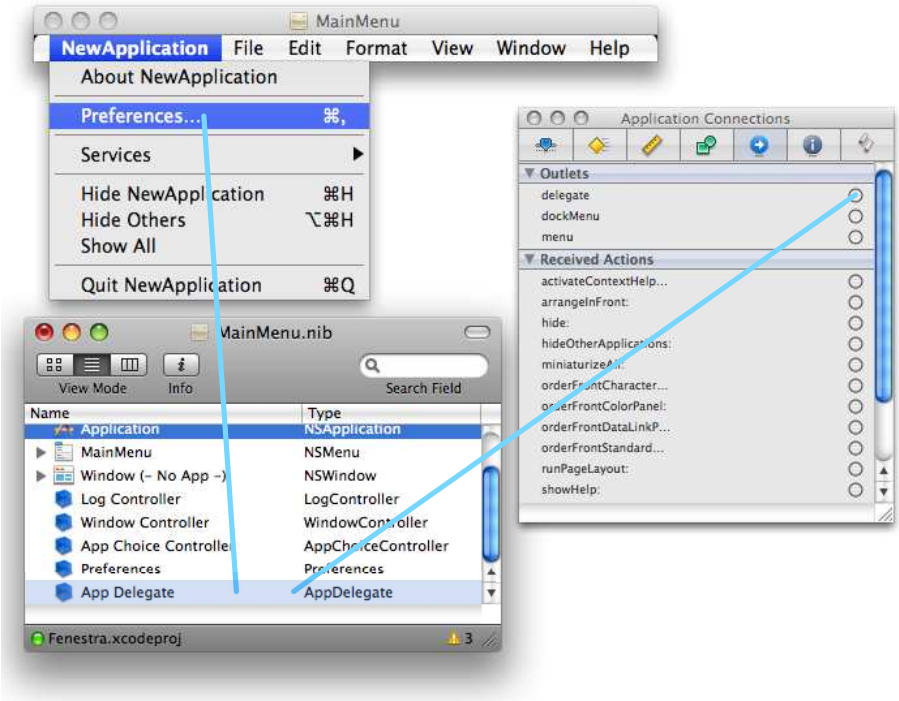
The - image is named NSRemoveTemplate.

12.3 Hooking the Panel to the App

The panel needs to be connected into the main program. Specifically, the Fenestra > Preferences menu item has to point at an action method that loads and launches the new nib.

Where should that method live? A typical place is in the application delegate. We used one of those in the status bar app, way back in Section 2.1, *A Program That Prints*, on page 23, but we haven't needed one in Fenestra. You'll have to create it (in MainMenu.nib, not Preferences.xib,

since the former contains Fenestra's menu bar). The application delegate will hook into the rest of the nib like this:



Create it by doing the following:

1. Drag an NSObject into the doc window.
2. Reclassify it as an AppDelegate object in the Identity inspector.
3. Give it a changePreferences action (also in the Identity inspector).
4. Connect the New Application[1] > Preferences menu item to changePreferences. As with a real menu bar, you can look at submenus of IB's picture of it by clicking. Once at the Preferences menu item, the easy way to connect it to the action is by Control-clicking it and then dragging to the AppDelegate entry in the doc window. When you release the mouse, you'll be prompted for which action to connect to.
5. Make the delegate outlet of the application point to the new AppDelegate.

1. We'll rename "New Application" to "Fenestra" in Chapter 22, *Fit and Finish*, on page 287.

Here was my first implementation of AppDelegate:

Download fenestra/binding-text-field/app/AppDelegate.rb

```
class AppDelegate < OSX::NSObject
  include OSX

  def changePreferences(sender)
    puts "Will launch preference panel"
  end
end
```

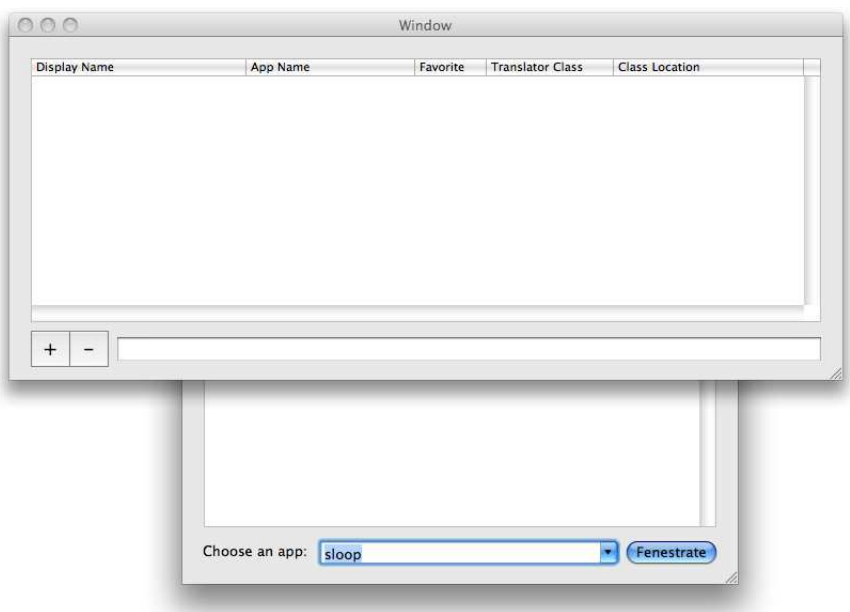
Amazingly enough, everything worked when I launched Fenestra and pressed Command-, to activate the Preferences menu item. (I usually mess up at least one step.)

Now we'll create the contents of the method. It has to load the nib file. One easy way to do that is to use NSWindowController, which has a method just for that:

Download fenestra/binding-text-field/app/AppDelegate.rb

```
def changePreferences(sender)
  wc = NSWindowController.alloc.initWithWindowNibName("Preferences")
  wc.showWindow(self)
end
```

When I run that, it *appears* to work, but look closely:



Although the preference panel is nicely on top of the main window, the main window's combo box's text field is highlighted, and the Fenestrate button is solidly blue, indicating that Return will press it. Apparently the main window is still the *key window* (the one keypresses go to).

All windows have a method, makeKeyWindow, that does what we want, and NSWindowControllers have an accessor, window, that lets us send that message to our panel. So, this should work:

Download fenestra/binding-text-field/app/AppDelegate.rb

```
def changePreferences(sender)
  wc = NSWindowController.alloc.initWithWindowNibName("Preferences")
  wc.showWindow(self)
  wc.window.makeKeyWindow
end
```

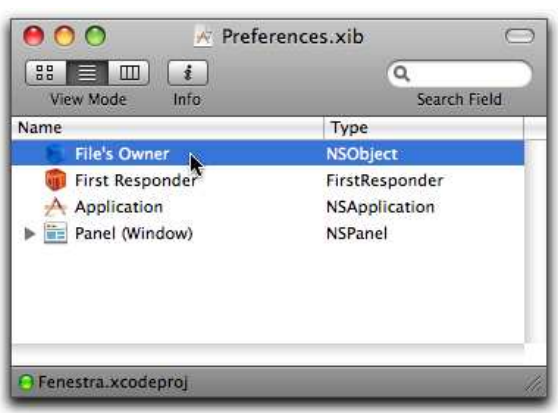
But it doesn't:

```
2008-09-06 09:58:44.163 Fenestra[22153:10b] AppDelegate#changePreferen⟷
ces: NoMethodError: undefined method `makeKeyWindow' for nil:NilClass
  ... /binding-text-field/app/AppDelegate.rb:17:in `changePreferences'
```

What's up?

12.4 The Nib File's Owner

The reason is something I've carefully avoided explaining until now. What's the meaning of "File's Owner" in the following?



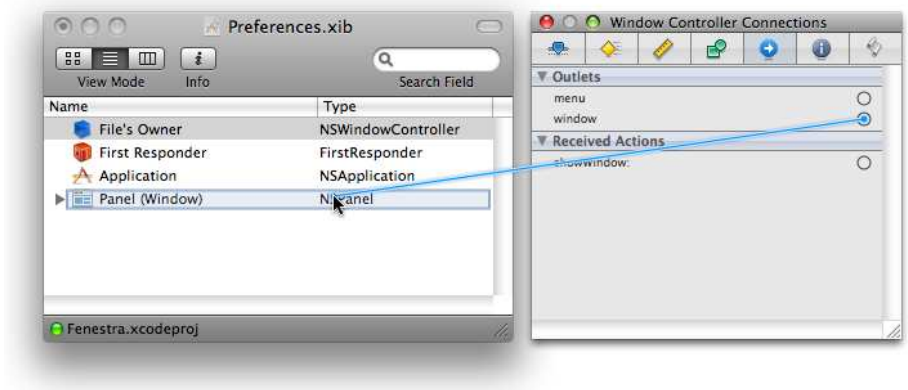
The File's Owner is the object that's responsible for all the objects in a nib file, usually the object that loaded it. It's one of two gateways nib objects have to the larger application. (The other is the singleton NSApplication object.)

In this case, the File's Owner is our NSWindowController. As it turns out, although it loads the nib file and creates the NSPanel, it doesn't connect its window outlet to that window. (I suppose that's because a nib file could have many windows in it.) We have to make the connection in the nib. That takes two steps:

1. All IB knows about the File's Owner is that it's some sort of NSObject. We have to tell IB it's specifically an NSWindowController (using the Identity inspector).

2. Now IB knows what kind of outlets the File's Owner has, so you can connect its window to the NSPanel.

That looks like this:



And Fenestra now works. (I won't show it, because black-and-white print obscures the change.)

12.5 IB's First Responder Pseudo-Object

If you look carefully, you'll notice that the table in the preference panel has a thin blue ring around it, indicating that it's the first responder—the control that gets keyboard and mouse events. Change that to be the text field, as you did for an earlier window in Section 5.4, *The Initial First Responder*, on page 80.

That change gives me an excuse to explain the last unexplained entry in the doc window: the one named "First Responder."

As a user works with a window—for example, by tabbing through it—the first responder will change. If an object wants to send a message to the current first responder, it uses an outlet to the "pseudo-object" First

Responder. You can see the list of messages a first responder should reply to in the First Responder's Connections tab. They include ones like paste, selectAll, saveDocument, and record. You can add ones of your own in the Identity tab.

12.6 Memory Leaks

Here, again, is changePreferences:

Download fenestra/binding-text-field/app/AppDelegate.rb

```
  def changePreferences(sender)
    puts "Will launch preference panel"
    wc = NSWindowController.alloc.initWithWindowNibName("Preferences")
    wc.showWindow(self)
    wc.window.makeKeyWindow
  end
end
```

It still has two problems, one silly and one serious:

- What happens if the user asks for the preference panel when it's already open? The code as is will create another NSWindowController with another window in it. It seems likelier that the user would rather the existing preference panel were raised to the top and made the key window.
- The nib objects are almost completely self-contained. The only "outside" object they can point at is the File's Owner, the NSWindowController. After our method returns, nothing points at the NSWindowController, so it and the nib objects are completely cut off from all the rest of the objects in the app. When the Ruby garbage collector runs, it will think they're garbage and reclaim their space. The resulting symptom is that the preference panel will suddenly vanish.

This version of changePreferences solves both problems:

Download fenestra/binding-hash/app/AppDelegate.rb

```
def changePreferences(sender)
  unless @wc
    @wc = NSWindowController.alloc.initWithWindowNibName("Preferences")
  end
  @wc.showWindow(self)
  @wc.window.makeKeyWindow
end
```

Using an instance variable instead of a local means that AppDelegate protects the NSWindowController and thus everything else in the nib file from the garbage collector. (The NSWindowController isn't garbage until the AppDelegate becomes garbage, and the AppDelegate never does.)

The **unless** makes changePreferences reuse the NSWindowController if it has already been created.

12.7 What Now?

Next we'll populate UI controls with data using the absolute minimum amount of code and the magic of Cocoa bindings. After that, we'll do the same but with the absolute maximum amount of code. That'll move you closer to understanding how bindings work.

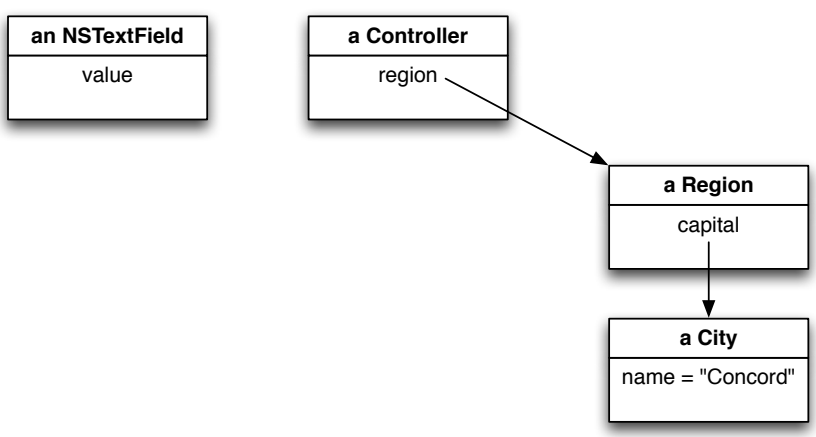
Chapter 13

Implementing a Preference Panel with Cocoa Bindings

In this chapter, we'll bring the preference panel to life. We'll do almost all of the work inside IB, though we'll have to write a little code.

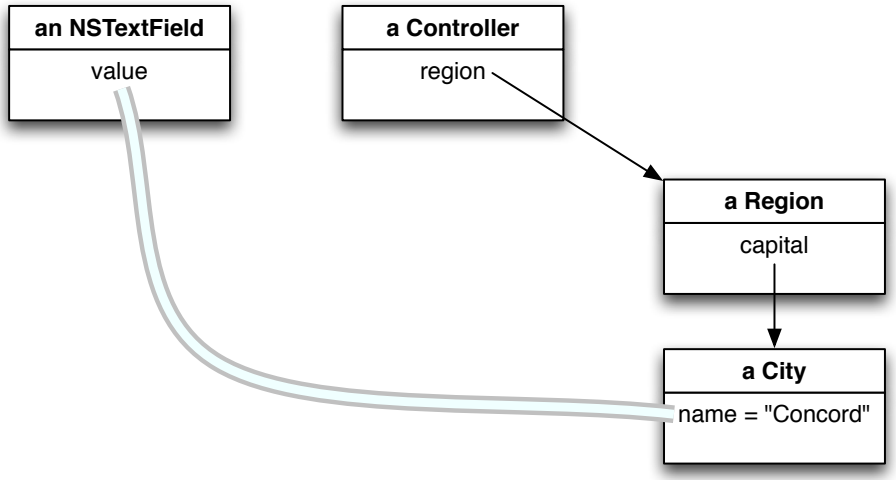
13.1 Binding a Simple Value

Consider this example:

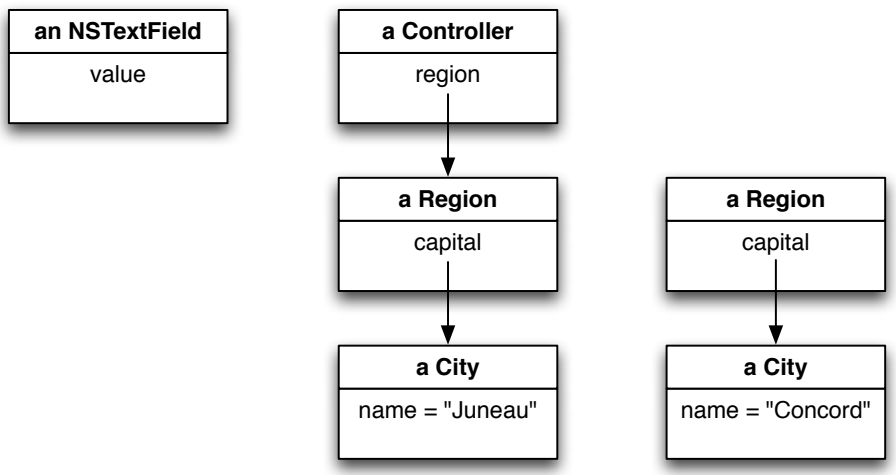


Suppose you want any change to the City's name to be reflected into the NSTextField's value, thereby changing the characters on the screen. That's a flow of data in one direction—from the app's interior to its surface. Suppose you also want more: if the user types something in the text field and completes that edit with Return or Tab (thus changing the value), you want the City's name to change.

In Cocoa, you'd say you want the NSTextField's value property to be *bound* to the City's name. Let's draw that by adding a double line that suggests a pipeline through which values flow back and forth:

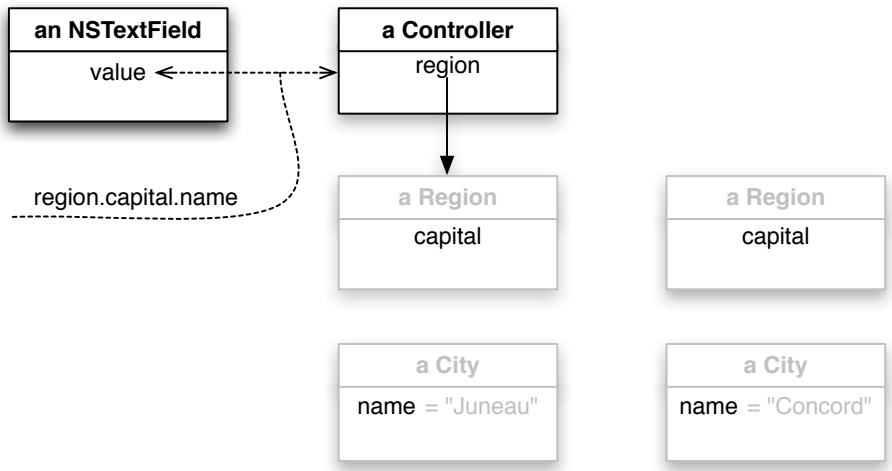


But that's likely not enough for you. What if the City's name stayed the same, but the Controller is changed so that it points to a new region (and thus a different City with a different name)? The result of such a change looks like this:



You'd still want the text field to change, in this case becoming "Juneau." That can be done by binding the NSTextField's value to the Controller's "region.capital.name" *keypath*.

How to draw that? A simple pipeline between objects will not work because it doesn't take account of objects along the keypath. Moreover, a keypath isn't a list of objects; it's a description of how to get from one object to another, which is a description that applies even if individual objects are swapped out for others. So, I'll draw a binding like this:



The two-headed arrow makes a connection between the bound property and the object *starting* the keypath. Thereafter, what matters is the keypath. Bindings work as if Cocoa traces along the keypath at each "tick" of the app's event loop.[1] It starts at the pointed-at object (usually a controller), follows the keypath from there, and records the identity of each object along the way, building a trace record that might look like this:

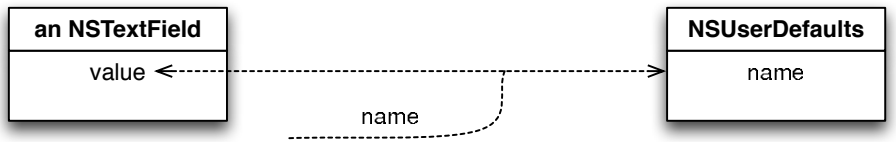
```
#<Region: id=0x1abf0b0>
#<City: id=0x1ac33c0>
#<NSCFString "Spfl": id=0x2ad834>
```

It then compares that trace to the one it made last time. If any of the object identities have changed, the bound object (in this case, the NSTextField) is informed. I'm going to defer explaining what happens then until Chapter 27, *The Underpinnings of Cocoa Bindings*, on page 350. Apple supplies ready-made objects that we can use in Fenestra. Once you've used such objects, it'll be easier to understand how to write them.

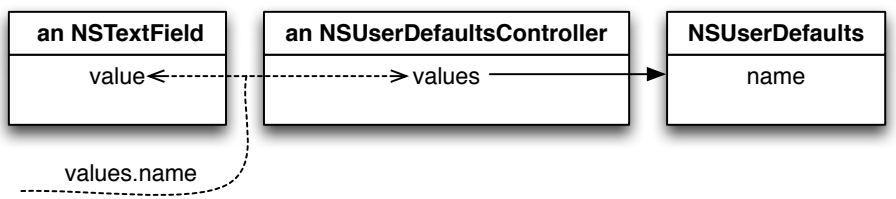
1. The actual mechanism is much more clever, but understanding it requires knowing a lot about how Objective-C is implemented.

The NSUserDefaultsController

An easy way to start using bindings in Fenestra is to bind the preference panel's text field to a new key—call it name—in the user defaults. It would seem that we could establish this binding:



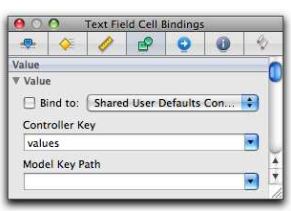
But that doesn't work because NSUserDefaults doesn't implement the methods that bindings require. (See Chapter 27, *The Underpinnings of Cocoa Bindings*, on page 350.) To bind user defaults, you need to put an adapter between the view and the NSUserDefaults:



The view binds to the NSUserDefaultsController using the keypath "values.name."

Let's make the binding in IB. Continue working with your latest version of Fenestra or, if you don't have one, the one in fenestra/binding-text-field. (If you use the latter, though, you'll lose the fix explained in Section 12.6, *Memory Leaks*, on page 160 unless you manually apply it.)

Select the text field at the bottom of the panel, and navigate to the Bindings inspector (the fourth inspector tab).[2] You'll see the following (plus more that I'll ignore for now) after you expand the value disclosure triangle:



2. You can select a control via its image in the preference panel or by expanding the NSPanel's entry in the doc window. On my machine, selecting a control via the doc window leads to an incredibly annoying IB bug where the inspector forgets what object you're inspecting as soon as you change something about bindings.

Notice that IB thinks the most likely object you want to bind to is an NSUserDefaultsController. That's perceptive of it. Click the Bind to checkbox. Notice that IB immediately creates an NSUserDefaultsController in the doc window.

Because Apple assumes you will always want some sort of controller object between the view and the data you want to bind, IB divides the keypath into two parts: one for the controller's property and the other for the rest of the keypath. Once again, it makes a good guess for the controller's component of the keypath: values is what you want. Make the Model Key Path "name." IB supplies the period between "values" and "name."

Save the nib file, run Fenestra, and bring up the preference panel. You should see an empty text field:

An NSUserDefaults object returns nil for a nonexistent key. It's nice of the text field not to choke on that.

Now enter some text in the text field, and press Return. Quit and restart Fenestra. When you bring up the preference panel, you should see this:

Presto! The value has changed. This example doesn't show that the binding is bidirectional; we'll see that in Section 13.4, *Transforming for Real*, on page 180.

Since the text field was just to demonstrate how easy a simple case is, we're done with it now. Delete it from the nib file.

13.2 Binding an Array of Hashes

Now we'll start filling table columns.

I'll avoid swamping you with too many facts at once by binding the table to hashes instead of TranslatorPreference objects. The hashes will be initialized in the already-existing init method, as shown in the marked lines ❶ and ❷ on the next page.

Download fenestra/binding-hash/app/preferences/Preferences.rb

```
def init
  only = TranslatorPreference.alloc.initWithHash(
              :display_name => "sample webapp com.exampler.counting",
              :app_name => "com.exampler.counting",
              :class_name => 'CountingApp',
              :favorite => true,
              :source => nil)
❶ hashprefs = [{ :display_name => only.display_name,
                 :app_name => only.app_name,
                 :favorite => false},
               { :display_name => "wevouchfor members",
                 :app_name => "org.wevouchfor.mem",
                 :favorite => true }
              ]

  @defaults = NSUserDefaults.standardUserDefaults
  @defaults.registerDefaults(:translators => collect_archived([only]),
❷                            :hashprefs => hashprefs)
  super_init
end
```

I've put only three columns' worth of data in the hash. Before we worry about the other columns, we'll have switched to using TranslatorPreference objects.

We can visualize the value of the key :hash_prefs as an array of key-value objects. (See the first part of Figure 13.1, on the following page. In it, array elements are stacked vertically.)

We'll have to adapt that to fit NSTable's way of organizing tables: by column. It's best not to think of a table as a two-dimensional array. Since you can drag columns around, column indices aren't guaranteed to stay the same. Instead, each column has a name and an array of values.[3]

So, our job is to transform the first part of Figure 13.1, on the next page, into the second, creating (in essence) a hash of arrays out of an array of hashes by taking a vertical slice out of the first structure and dropping the resulting array into the second. That would be easy to do in Ruby:

```
table[:display_name] = hashes.collect { | h | h[:display_name] }
```

3. Strictly, a column has two "names": its title (which you already set) and an identifier (which you can set if you like).

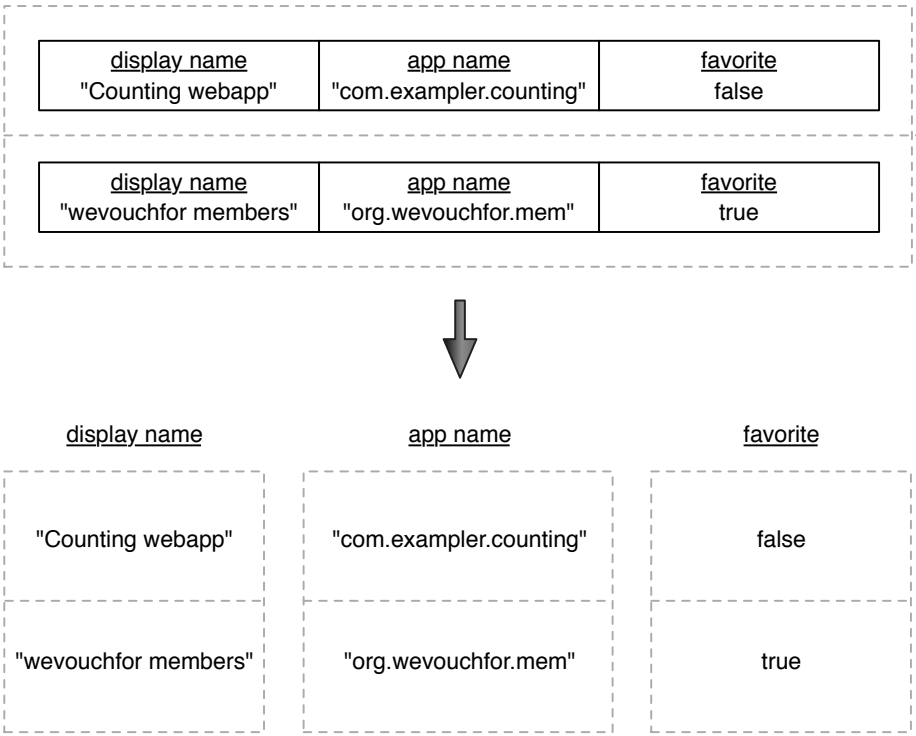
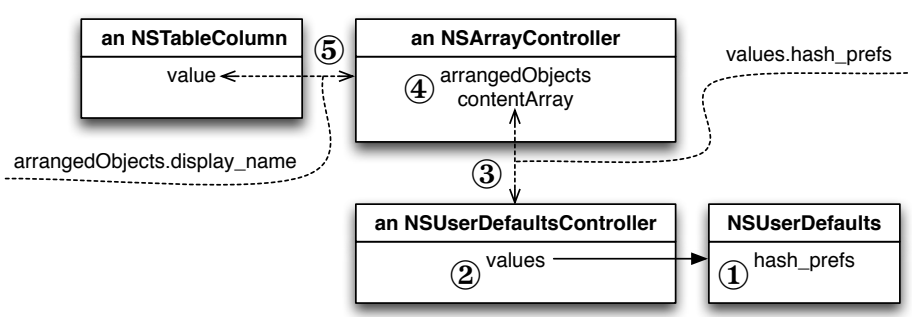


Figure 13.1: The transformation we need

Cocoa comes with a prepackaged controller that can do the same kind of thing: NSArrayController. It can be plugged in between a control and some data source that should be treated as one or more arrays. Here's the flow of data:



1. If asked for its hash_prefs key, the NSUserDefaults object returns an NSArray of NSDictionary objects (which RubyCocoa lets us treat like an array of hashes).
2. The NSUserDefaultsController adapts NSUserDefaults to make the hash_prefs value available via the NSUserDefaultsController's "values.hash_prefs" keypath.
3. NSArrayController's contentArray can be bound to that keypath.
4. An NSArrayController doesn't directly expose its content array. Instead, it exposes derivatives that are tuned for user interface operations. In our case, we'll use arrangedObjects. That's an array of the same objects, but if the user has chosen to sort one of the table columns (by clicking the column header), they're in sort order.

   Here's a way to picture these two arrays:

   The arrays refer to *the same objects*, just in a different order. Because of contentArray's binding, these objects are identical to the contents of the preference file.
5. Now an NSTableColumn's value property can be bound to the keypath "arrangedObjects.display_name." Because arrangedObjects is an array, the remainder of the keypath is handled as if with collect. That is, the result is equivalent to this:

```
arrangedObjects.collect { | hash | hash[:display_name] }
```

In effect, what we've done is create a pile of different aliases for the same display_name:

- aTableColumn[0] will always be the same as. . .
- . . . anArrayController.arrangedObjects[0][:display_name], which will always be the same as. . .
- . . . anArrayController.contentArray[2][:display_name], which will always be the same as. . .
- . . . anNSUserDefaultsController.values[:hash_prefs][2][:display_name], which will always be the same as. . .

- . . . anNSUserDefaults.objectForKey(:hash_prefs)[2][:display_name], which will always be the same as. . .
- . . . what's stored in the preference file.

Easier to Do Than to Describe

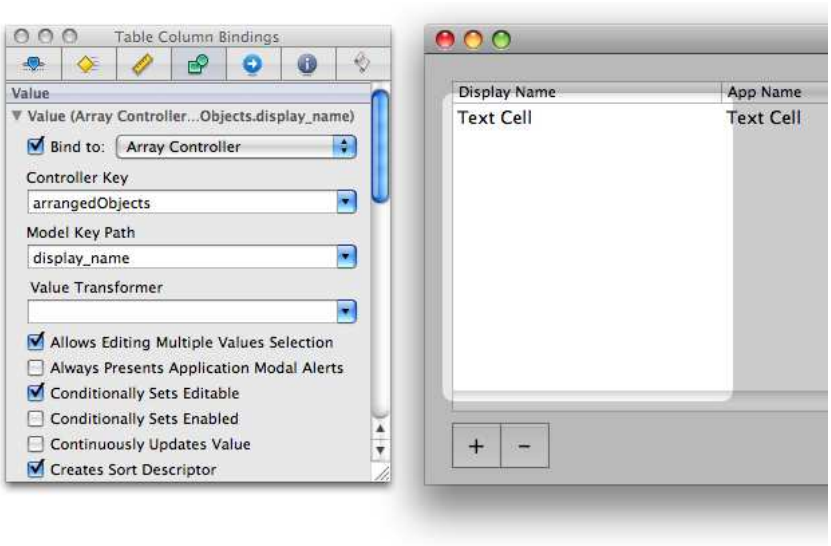
To create these bindings, do this in IB:

1. Drag an NSArrayController from the library into the doc window.
2. Make its Bindings tab in the inspector look like this:

   In particular:

   a) Select the "Bind to" checkbox.
   b) Make sure it's bound to the Shared User Defaults Controller.
   c) Check that the controller key is values.
   d) Set the model keypath to hashprefs.
   e) Make sure Handles Content as Compound Value is selected, or else the NSArrayController will not follow the path into the hash. (I'll explain in more detail in the sidebar on page 195.)

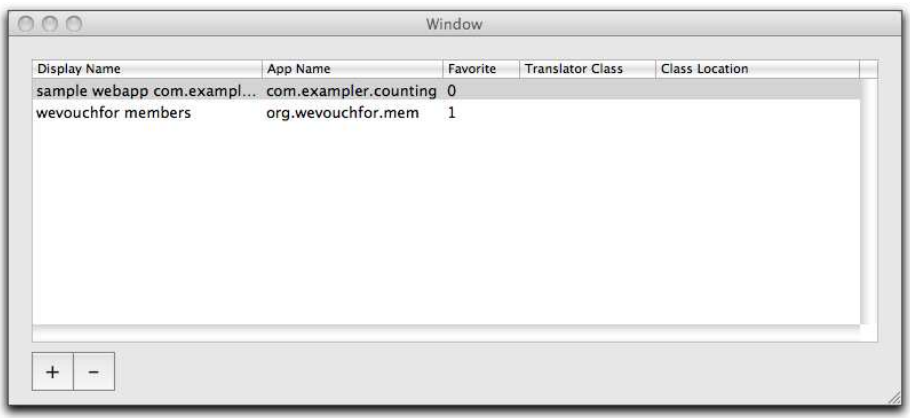
   Check your work! I keep having to solve mysteries because I'm careless. The downside of not having to write code is you have no place to put debug statements (at least for now; see Section 13.4, *Snooper*, on page 178).
3. Make the Display Name column's binding look like this:



The column should be bound to the NSArrayController with the full keypath "arrangedObjects.display_name."

4. Do the same for the App Name and Favorite columns.

Now save, build, and run. You should see this preference panel:



Notice that the Favorite column displays integers. That's because it's bound to an array of Cocoa booleans—that is, to integers. We'll deal with that in the next section.

Make a change to one of the fields (except for Favorite), exit Fenestra, restart it, go to the preference panel, and see whether the change was saved. (Note: the change won't appear in the main window's combo box because that uses :translators, not :hash_prefs. The two windows will

start using the same name in Section 13.4, *Value Transformers*, on page 177.)

13.3 Formatters

Let's use Cocoa *formatters* to clean up the Favorite column. A formatter is used to convert an object into text for a cell, and vice versa. In our case, we need to convert the Cocoa boolean values #<NSCFBoolean true> and #<NSCFBoolean false> (that is, 1 and 0) into the strings *"yes"* and *"no"*. And we need to convert those strings back into booleans should the user change them.

Converting a boolean into a string is straightforward:

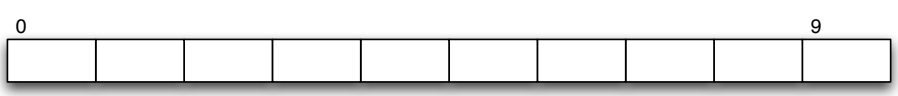
Download fenestra/binding-formatter/app/preferences/formatters.rb

```
class BooleanCellFormatter < OSX::NSFormatter
  def stringForObjectValue(o)
    o == true.to_ns ? "yes" : "no"
  end
end
```

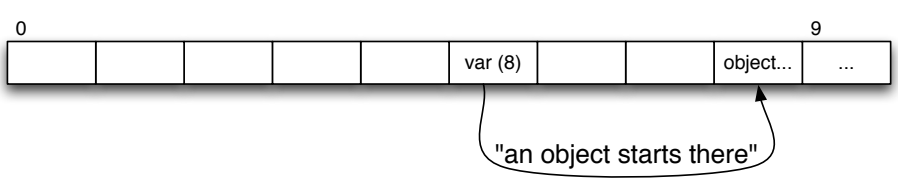
Going in the other direction is a little more difficult. Before I show you how, I have to make a little speech about memory.

Pointers to Pointers

As far as the chip inside your computer knows, there are no objects; there's just a huge array of memory divided into slots numbered from zero to whatever:



Objects are just a consensual hallucination we have, one that the Ruby and Objective-C runtimes support. Those runtimes translate our language about "variables" and the "objects" they "name" (or "contain" or "point to") into this:

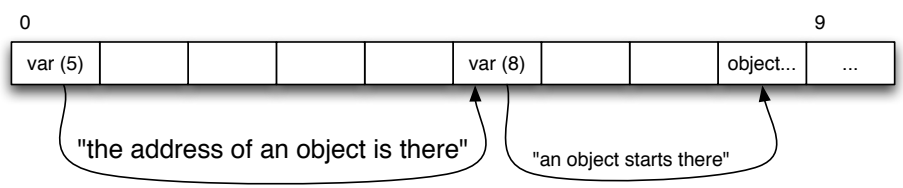


When we say "variable," the Ruby runtime knows we really mean one memory slot that contains the address (index) of another one. For example, the variable at slot 5 contains the number 8, to be interpreted as an address (rather than, say, a number to add). Because the actual values (5, 8, and so on) are really irrelevant and people are good at following arrows, we usually talk of *pointers* rather than addresses or indices.

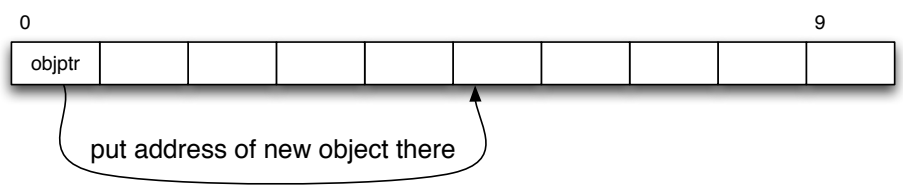
Ruby further knows that the memory slots starting at the pointed-to address follow a stylized layout that fits what humans mean when they say the word *object*. For example, it knows which of the memory slots in the object points to its class (yet another object).

In Ruby, that's all there is: memory slots that point to objects. Because of that, we can almost always ignore the difference between the address of an object and the slots that comprise it. We can say "the object returned by the find method" when we really mean that find returns the *address* of an object.[4]

Objective-C, though, can take things further. A variable can point to another variable:

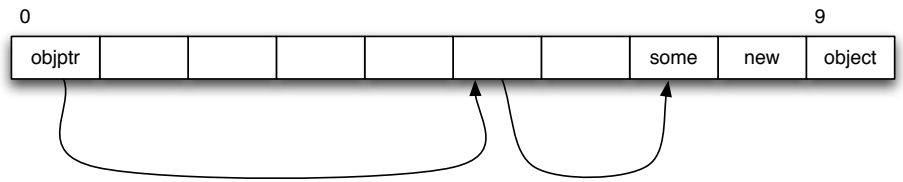


Objective-C programmers use these when they want a method to return multiple values. The method for converting a cell text into an object needs to return three values: a boolean, a string, and some just-created object. Because Objective-C doesn't allow multiple return values, the calling method has to pass two pointer-to-pointer arguments. Here's the setup for the last argument at the moment the method is called:



4. The time you can't ignore the difference is when several variables point to a single object. If you say "The result is the string *"foo"*," you have to remember that the contents of the string can still be changed via some other variable.

The called method makes an object, which means asking the runtime for a pointer to some previously unused space, and then stuffs that pointer into the slot the pointer-to-pointer argument points at:



Now the calling method can follow the pointer-to-pointer to retrieve the (pointer to the) object.

Ruby has no notion of pointers to pointers. So, RubyCocoa converts such an argument into an ObjcPtr. To "return" an object through it, you set one of its instance variables to the object. Then, as the method returns into the Objective-C universe, RubyCocoa converts the ObjcPtr back into a pointer-to-pointer.

What does that look like in code? Like this:

Download fenestra/binding-formatter/app/preferences/formatters.rb

```
def getObjectValue_forString_errorDescription(objptr, s, errdesc)
  case s.to_ruby.downcase
    when 'yes': objptr.assign(true)
    when 'no': objptr.assign(false)
    else return false
  end
  true
end
```

(The errdesc argument, unused here, can be used to pass back error descriptions to callers who want them. Table cells don't, so they pass in nil. If they did, errdesc would be an ObjcPtr.)

How do you know when you need an ObjcPtrc? Cocoa's API declaration for our method shows the two indicators:

```
(BOOL) getObjectValue: (id *) anObject
       forString: (NSString *) string
       errorDescription: (NSString **) error
```

To explain them, let me start with (NSString *), which is used for the string (second) argument. That's the Objective-C declaration of what, in Ruby, we'd call an NSString argument. Read the asterisk as "pointer to"—Objective-C reveals the pointerness at the heart of the object world, whereas Ruby hides it.

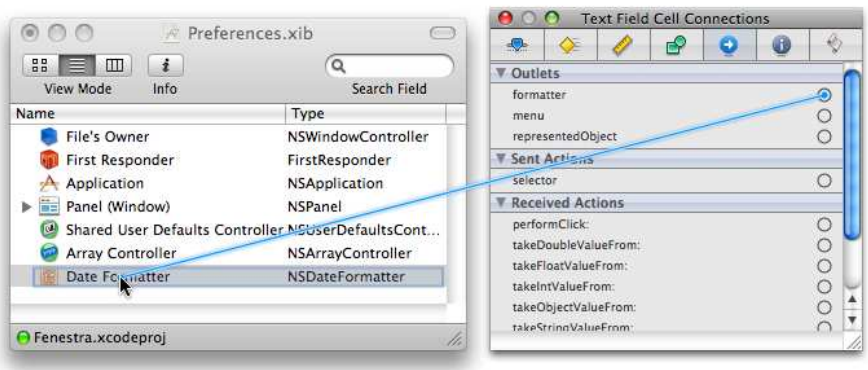
If one asterisk means a pointer, then two asterisks mean a pointer to a pointer. So, the (NSString **) declaration for error tells you to expect an ObjcPtr that you'll assign an NSString.

As far as a RubyCocoa programmer is concerned, the word *id* is shorthand for NSObject *, so (id *) also means you need an ObjcPtr.

Underneath its declaration, the method's description is likely to say a variable "returns by reference" or "returns by indirection."

Connecting the Formatter

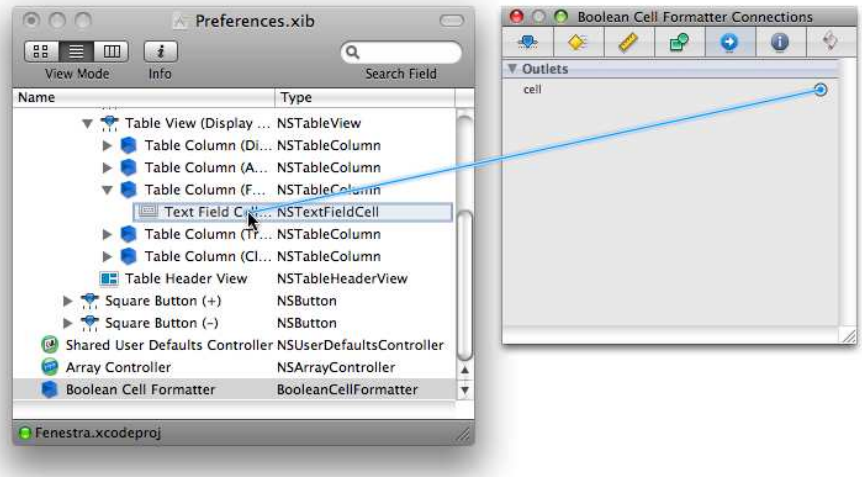
IB comes with two predefined formatters, one for numbers and one for dates. Connecting them to a text cell is easy: drag one of them in from IB's Library, and then connect the text field's formatter outlet to it.



But IB doesn't know about our formatter, which causes an annoyance. When we create one in IB, we do it in the now-familiar way: drag an NSObject from the Library and use the Identity inspector to change its class. That's fine, but IB then refuses to make a connection between a text cell's formatter and our object. Probably it's being too smart for our good: since *it* doesn't know that the object is an NSFormatter, *we* must be making a mistake.

Fortunately, we can work around IB's smug assertion of superior knowledge with our powers of hackishness. First, give BooleanCellFormatter a cell outlet, and connect that to the text cell, making an outlet that flows backward.

Like this:



Then have the formatter's awakeFromNib use the initialized outlet to set the text cell's formatter:

Download fenestra/binding-formatter/app/preferences/formatters.rb

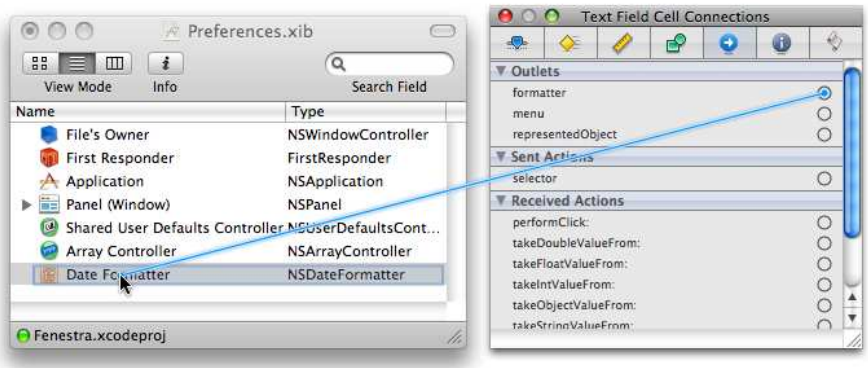
```
class BooleanCellFormatter < OSX::NSFormatter
  ib_outlet :cell

  def awakeFromNib
    @cell.formatter = self
  end
  #...
end
```

That done, Fenestra will print Favorite values nicely. Try it and see.

Try This Yourself

If you try to set Favorite to some value other than "yes" or "no," you'll get a pop-up forbidding it. It would be better if Fenestra prevented you from typing anything but the two valid values. If you type a wrong character, it should simply not echo. Use NSFormatter's isPartialString-Valid:newEditingString:errorDescription method to accomplish that. (Hint: you don't need to do anything with the last two arguments.)

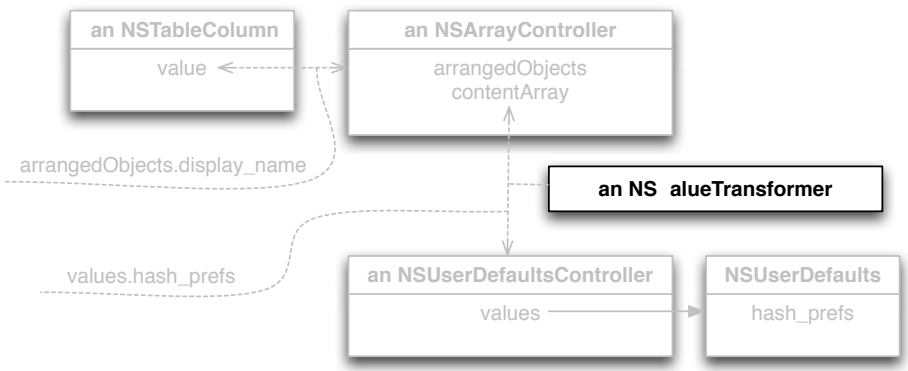
You can find my solution, commented out, in fenestra/binding-formatter/app/preferences/formatters.rb. Connecting them to a text cell is easy: drag one of them in from IB's Library, and then connect the text field's formatter outlet to it:



13.4 Value Transformers

Now let's use TransformerPreference objects instead of hashes. The problem here is that TransformerPreference objects are stored as NSData inside the preferences file. Our earlier Preferences class handled archiving and unarchiving them. Somehow, we have to make the binding controllers do the same.

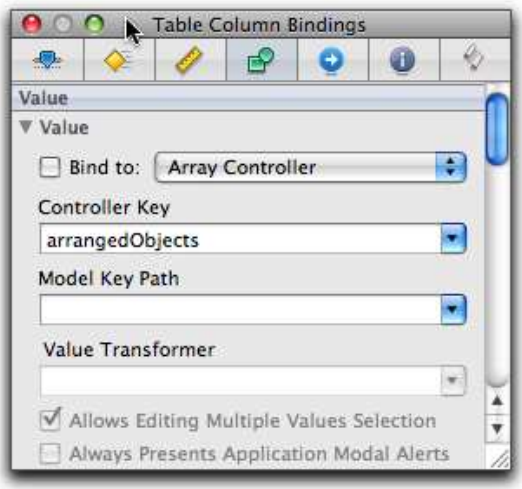
Fortunately, we can attach a *value transformer* to each binding. If we attach a value transformer to our NSArrayController's binding to NSUserDefaultsController, it will intervene in the flow of data between the two:



Snooper

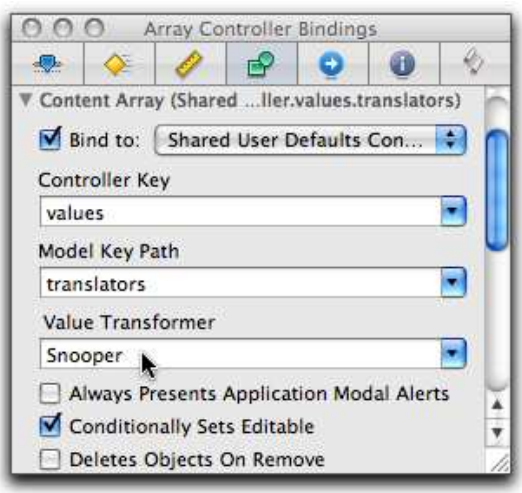
To see transformation in action, let's begin with a value transformer that does nothing but print the values it receives. Do the following:

1. Disconnect the table columns from the array controller:



   If you don't, you'll get confusing error messages when the table tries to use keys that we haven't set up yet.

2. Enter Snooper into the Value Transformer field of the array controller's binding. Also change the Model Key Path setting to translators instead of hashprefs:



3. Implement Snooper. Because it's a handy class for debugging, I put it in app/util:

Download fenestra/binding-transformer/app/util/value-transformers.rb

```
class Snooper < OSX::NSValueTransformer
  include OSX

❶   def transformedValue(value)
      NSLog "Transforming: #{value.inspect}"
      value
    end

❷   def reverseTransformedValue(value)
      NSLog "Reverse transforming: #{value.inspect}"
      value
    end

❸   def self.allowsReverseTransformation; true; end


end
```

❶ The transformedValue method normally converts values that come from inside the program into something more palatable to classes closer to the user interface. Snooper just prints them out and passes them along without conversion.

❸ If the user changes a value in the user interface, the reverseTransformedValue converts it into a format preferred by interior classes.

❸ If there's no way for the user to change a user interface value, there's no need for reverseTransformedValue. By default, a value transformer is one-way. Defining this method to return true signals that it's two-way.

If you run Fenestra, you should now see output like this when you bring up the preference panel:

```
Transforming: #<NSCFArray [#<OSX::NSCFData:0x20673c class='NSCFData' ↩
id=0x8b8000>]>
```

Transforming for Real

We now have to transform that array of NSData into an array of TransformerPreference objects. Since we already do that in Preferences, it's easy enough to do again:[5]

Download fenestra/binding-transformer/app/preferences/value-transformers.rb

```
class DataArrayTransformer < OSX::NSFormatter
  # ...
  def transformedValue(nsdata_array)
    nsdata_array.collect do | datum |
      NSKeyedUnarchiver.unarchiveObjectWithData(datum)
    end
  end

  def reverseTransformedValue(pref_array)
    pref_array.collect do | pref |
      NSKeyedArchiver.archivedDataWithRootObject(pref)
    end
  end
end
```

After creating this code, make it the value transformer for the array controller. This time, connect all the columns.

Run the code, and you should see something like the following (depending on what you have in the preferences file):

5. We'll clean up the duplication later.

Change the favorite, exit the program, and restart it. Did the change stick? (Note that we don't have any code yet that restricts you to one favorite.)

Change the display name of an entry. Without exiting the preference panel, drop down the main window's combo box list: does your new name appear?

Make sure you can see the preference panel and then fenestrate a new app by typing text into the combo box and clicking the Fenestrate button. What happens? Is that not cool?

Try This Yourself

Edit BooleanCellFormatter to log the value that's formatted:

Download fenestra/binding-transformer/app/preferences/formatters.rb

```
def stringForObjectValue(o)
  NSLog "Favorite is #{o.inspect}."
  o == true.to_ns ? "yes" : "no"
end
```

Bring up the preference panel. Notice anything odd?

I noticed that the first value logged doesn't seem to come from the preferences file but rather from the nib. . . or something:

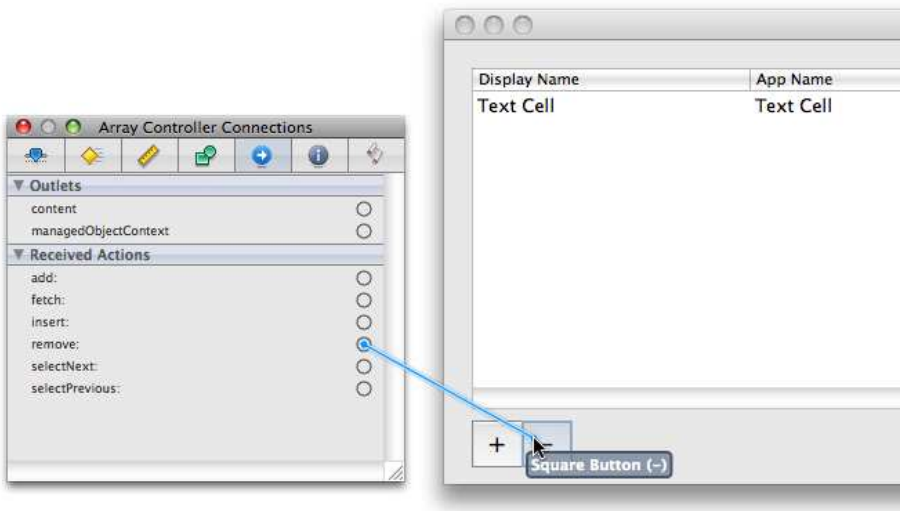
```
Favorite is #<NSCFString "Text Cell">.
Favorite is #<NSCFBoolean false>.
```

I can't account for that. I changed stringForObject to return the NSCFString value when it's given (instead of *"yes"* or *"no"*). It never appears in the table.

So, all along this particular formatter has been spuriously but harmlessly returning "yes" the first time it's called. A different formatter might choke if given a string when it expects something else. You have been warned.

13.5 Adding and Deleting Table Rows

Array controllers are built already knowing how to add and delete table rows, so it's easy to make the buttons work. First, bind the – button to the array controller's remove action:



Do the same with the add action.

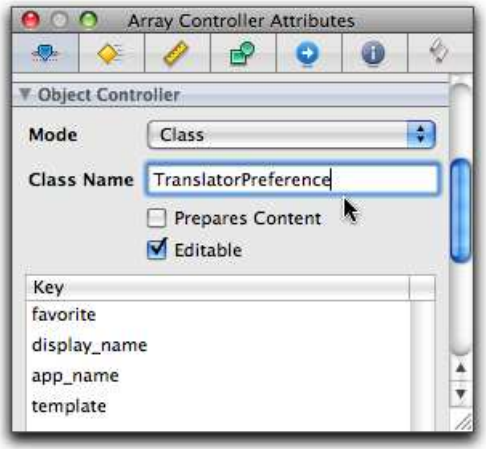
By default, add creates the new datum like this:

```
NSDictionary.alloc.init
```

You need to tell it to do this:

```
TranslatorPreference.alloc.init
```

You do that by changing the class name of the controlled object in the Attributes inspector:



We also need to give TranslatorPreference an init method. Otherwise, the newly created TranslatorPreference's instance variables will all be nil. The combo box will blow up when it tries to use a nil display_name as an entry in its pop-up list. Here's such an init method:

Download fenestra/binding-transformer/app/preferences/TranslatorPreference.rb

```
def init
  initWithHash(:display_name => 'Display name',
               :app_name => 'App Name',
               :class_name => 'ToString',
               :favorite => 'false',
               :source => nil)
end
```

The result of clicking the add button still isn't ideal, though:

The new line should be highlighted, the cursor should be in the first columm of the new row, and that cell should be selected for editing. We'll fix those things in Chapter 16, *Selections and Editing*, on page 202.

13.6 What Now?

I'm one of those creaky old-timers who's suspicious of graphical tools that aim to replace code. I'm suspicious for two reasons: cases where I couldn't get the tool to do what I wanted and cases where I suffered because I didn't understand what was really happening behind the scenes.

In writing this book, I found both things were true—in a mild way—of Interface Builder. I had to write a lot of test code to figure out how bindings really work. That turned out not to be too much of a problem—with one exception, you don't need to know more than you know now to finish Fenestra—so I've pushed the gory details into the reference chapters (at Chapter 27, *The Underpinnings of Cocoa Bindings*, on page 350). Read them at your leisure.

But not being able to do something I needed proved to be a bigger problem. For selfish reasons that I'll explain later, I wanted to replace bindings programmatically. That means *making* them programmatically—and that means explaining how to do that to you. That's what the next chapter does.

Chapter 14

Setting Up Bindings with Code

In the previous chapter, you declared what bindings you wanted by editing the nib file. The nib-loading process did the work of binding objects together. In this chapter, you'll learn how to make bindings with Ruby code.

Fenestra's current organization is shown in Figure 14.1, on the next page. We'll replace the numbered NSArrayController with our own object. That'll be a subclass of NSArrayController that both creates and binds to the NSDefaultsController below it. But first...

14.1 Oh No! Terminology!

If you browse around the Web, you'll see that a lot of people find Cocoa bindings hard to understand. I certainly did. The question is, why? I've decided the root cause is statements like the following, from Apple's *Cocoa Bindings Programming Topics* [App08f]:

> A binding is an attribute of one object that may be bound to a *property* in another such that a change in either one is reflected in the other.[1]

This statement, although technically true, misleads. It makes it easy to assume the following:

- That the binding mechanism is bidirectional. It's not. Each of the controls we bound in the previous chapter was specially coded to make its bindings *appear* bidirectional, but in fact the two objects

1. *Property* is the Objective-C word for what Ruby usually calls an *attribute*—think attr_reader. As you'll see later, when the Cocoa documentation refers to *attribute*, it doesn't necessarily mean a property that's accessible via getters and setters.

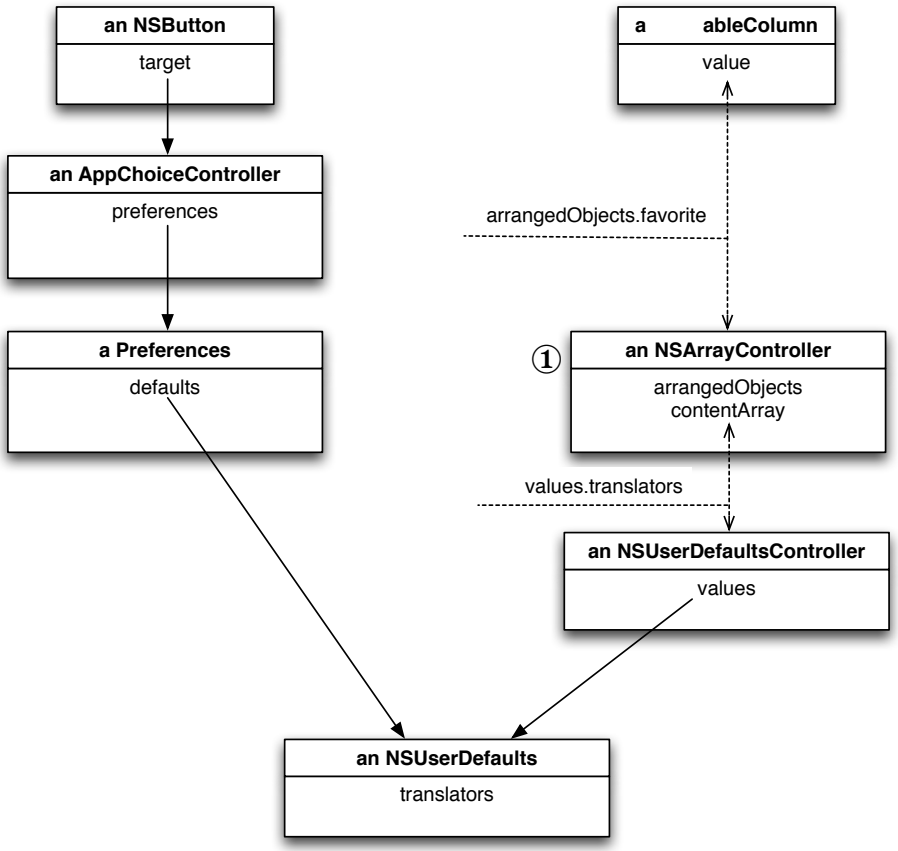


Figure 14.1: The starting structure and the first change

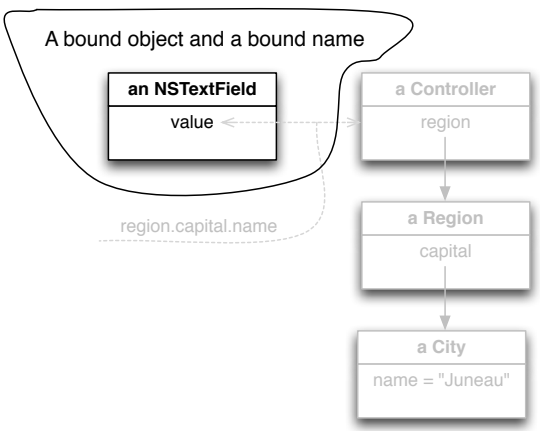
in a binding are not symmetrical. (For example, although you can bind a button's value to a translator's display name, you can't do the reverse.)

- That the "attribute" mentioned in the quote is the same kind of thing as the "property"—basically, some sort of instance variable whose value gets changed. It's not. It's better to think of it as an arbitrary name used by binding-support code.

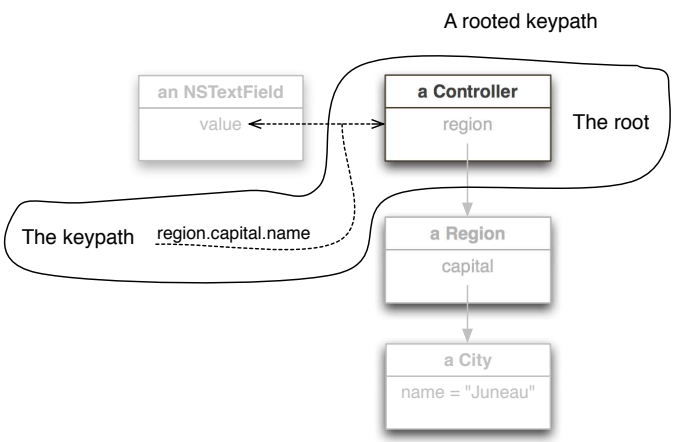
What I have decided is that explaining bindings requires two things. It requires an example that gradually moves from surface behavior down into the supporting technology. And it requires terminology that emphasizes how the two sides of a binding are different, not how they're similar.

The rest of this section gives the terminology I'll use. I wish I wasn't diverging from Apple's terminology, but I am.

In IB, you create a binding within a particular object's Bindings inspector. That object is the *bound object*. The specific "thing" that's bound is the *bound name* (or sometimes, if it hasn't been bound yet, the *binding name*). In the following picture, the bound object is an NSTextField, and the bound name is "value."

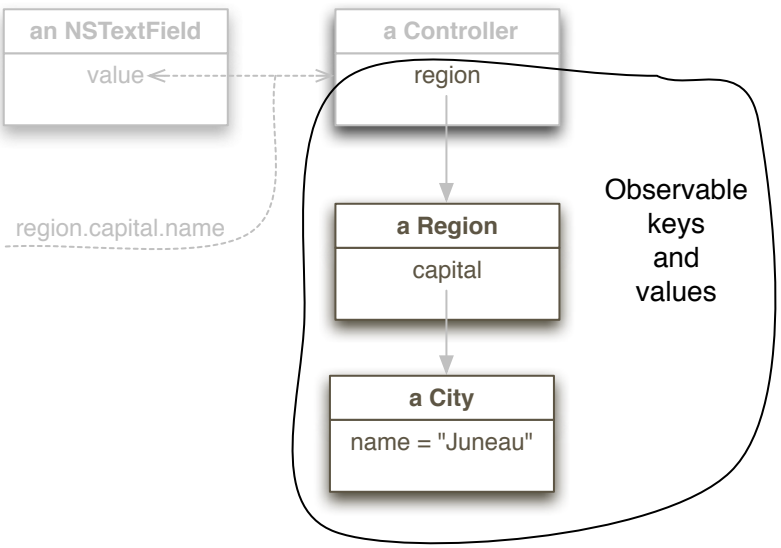


The bound object's bound name is bound to a *rooted keypath*. That's the combination of the object on the other end of the binding and the keypath Cocoa follows looking for changes to a property. If I need to talk about the object itself, I'll call it the *root*, and the keypath alone will just be *keypath*. In the next picture, the root is the Controller, and the keypath is "region.capital.name."



Each element of the keypath corresponds to an *observed property key*. I'll sometimes shorten the phrase to *observed key* or use *observable*

property key if the binding hasn't been made yet.[2] The word *key* should remind you of a hash or NSDictionary. That's good, since the technologies of binding try to make NSDictionary objects and objects with instance variables look alike from the outside. To reinforce the similarity, I'll say that each observed key is associated with an *observed property value*, or just *observed value*. In the following picture, the observed keys are "region," "capital," and "name," and the observed values are a Region, a City, and the string *"Juneau"*.



The end of the rooted keypath is special. The value there is the one that's used to update the bound object. In the previous picture, the NSTextField will be updated if the City changes its name. It will also be updated if any observed key anywhere earlier in the keypath is given a new value. That is, if the Region changes its capital, its capital's name is used to update the text field—even though that name didn't itself change.

2. The reason to use "observed" instead of "bound" is, first, to help avoid confusion between the two ends of a binding and, second, because the underlying technology is called *key-value observing*. See Section 27.3, *Declaring Observed Properties*, on page 352, and Section 27.4, *Observing Changes*, on page 353.

14.2 Using Rooted Keypaths in Code

In regular Ruby code, you traverse structures like the ones in the previous section using *boxcar notation*:[3]

```
controller.region.capital.name
```

Objects that can be part of a rooted keypath can be traversed with a different notation:

```
controller.valueForKeyPath('region.capital.name')
```

As you might guess, valueForKeyPath is part of the implementation of bindings; that connection is explained in Chapter 27, *The Underpinnings of Cocoa Bindings*, on page 350. I mention it here because I'll be using it throughout the next part of the book, *Fun with Tables*.

One nice feature of this notation is the way arrays are handled. Suppose we had an object country whose regions property was an array of Region objects. We could then get an array of all the capital names like this:

```
country.valueForKeyPath('regions.capital.name')
```

In effect, using an array in a keypath implicitly does a Ruby collect. This mechanism is the reason a table column can be bound to a keypath like "arrangedObjects.display_name" and can use the result to fill its cells.

14.3 Subclassing NSArrayController

We will begin by replacing the NSArrayController shown in Figure 14.1, on page 186, with a subclass, PreferencesController, that does *exactly* the same thing. That seems trivially easy, but I've found that changing bindings is like brain surgery: one little slip can lead to extremely odd behavior. So, we'll break the task down into even smaller steps, and I'll point out binding pitfalls along the way.

Replacing the NSArrayController

First, we replace NSArrayController but leave everything else unchanged. To do that, follow these steps. (You can examine my results in fenestra/binding-by-hand-1.)

3. So called because the property names are connected with periods the way boxcars in a train are connected with couplers.

1. Create this class:[4]

   Download fenestra/binding-by-hand-1/app/preferences/PreferencesController.rb

   ```
   class PreferencesController < OSX::NSArrayController
   end
   ```

2. In Interface Builder, create a PreferencesController in Preferences.xib by dragging an NSArrayController from the library and changing its class in the Identity inspector to PreferencesController. Do *not* drag in an NSObject (as we've always done before), because the Bindings inspector won't believe the new object has any of NSArrayController's binding names.

3. Still using IB, make the PreferencesController's Content Array binding look like NSArrayController's. That is:

   a) Bind it to the "values.translators" keypath in NSSharedUserDefaultsController. When doing so, I've sometimes gotten assertion failures from IB. They invited me to crash the app or ignore the failure. I've always ignored them, without harm.

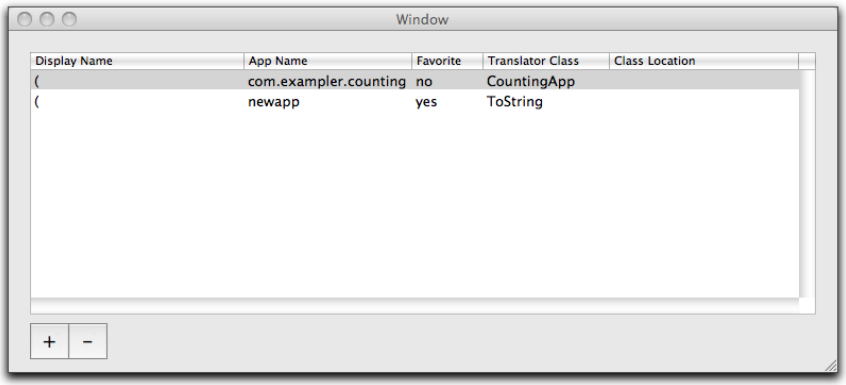
   b) Make the data transformer DataArrayTransformer.

   c) *Be sure* to select Handles Content as Compound Value. If you don't, the table columns will display correctly, but changes you make to them won't be saved. That's an easy problem to miss until long after you've forgotten about creating the PreferencesController, making it unduly hard to track down. (See the sidebar on page 195 for the details.)

4. Change the Value bindings in *all* the table columns to use the PreferencesController as the root object (instead of the NSArrayController). I emphasize "all" because when I first made these changes, I changed only the first column, Display Name. (Why change the rest until I saw whether that one worked?)

4. I put the new class in the preferences folder even though it would make just as much sense to put it in controllers. That suggests I've organized my files wrongly. Story of my life.

This was the result:



What's with the bogus parentheses in the first column? When I used the arrow keys on my keyboard to move around in one of those cells, I discovered Cocoa's printed representation of an NSArray. For some reason, each cell in the column gets the data for the whole column, not just the array element appropriate to its row number.

Binding all the columns to refer to the same controller fixes that. It took me rather a long time to figure that out, and it was not a happy time.

5. Run Fenestra to see whether it works. Try changing a display name in the preference panel and see whether it's reflected in the main window's combo box list. See whether fenestrating a new app reflects into the preference panel. See whether changes are retained after you quit Fenestra.

Hooking Up Preference Panel Buttons

The two buttons on the preference panel still point at the NSArrayController. So:

1. Add remove and add actions to the Identity tab of the PreferencesController.
2. Use the Attributes inspector to change the PreferencesController's Class Name attribute from NSMutableDictionary to TranslatorPreference. (The Class Name field can be found under the Object Controller disclosure triangle.) If you leave the class as is, some of your translators will end up as NSDictionary objects, which will blow up the DataArrayTransformer the next time you start Fenestra.

3. Connect each button to the appropriate action. It'll probably be easiest to Control-click the button and drag the connection to the action.
4. Delete the NSArrayController from Preferences.xib.
5. Run Fenestra to check your work.

14.4 bind_toObject_withKeyPath_options

We'll begin to make our subclass diverge from NSArrayController by having it programmatically create and bind to the NSUserDefaultsController, rather than having that work done in the nib file. The app's structure when running will still look like Figure 14.1, on page 186.

Here's that code:

Download fenestra/binding-by-hand-2/app/preferences/PreferencesController.rb

```
class PreferencesController < OSX::NSArrayController
  include OSX

❶   def initWithCoder(decoder)
      super_initWithCoder(decoder)
❷     self.objectClass = TranslatorPreference
❸     @defaults_controller = NSUserDefaultsController.sharedUserDefaultsController
❹     @transformer = DataArrayTransformer.alloc.init

❺     self.objc_send(:bind, 'contentArray',
                     :toObject, @defaults_controller,
                     :withKeyPath, 'values.translators',
❻                    :options, {
                       NSHandlesContentAsCompoundValueBindingOption => true,
                       NSValueTransformerBindingOption => @transformer
                     })
      self
    end

❼   def init
      NSLog "I should never be called."
      raise "Init should not be called in an NSArrayController subclass."
    end


end
```

❶ In Preferences, there's code that needs to run before the awakeFromNib method is called. I created an init method for it. (See Section 11.3, *Implementing the Adapter*, on page 141.)

Here, I'm defining initWithCoder instead. That's the same method used to unarchive TranslatorPreferences from the preference file. (See Section 11.2, *Archiving*, on page 134.) Why use it?

Consider an NSComboBox in IB's library. Although we (and Apple's documentation) refer to it as an object, it's actually a template for an object—once you create the object by dragging the template out of the library, you can customize it by setting any of a whole pile of attributes. Now, in object-oriented programming languages, that's part of the role of classes: they act as templates for their instances. And, indeed, IB library "objects" are really classes. Dragging one actually creates an object (with alloc and init). Selecting inspector checkboxes, typing into inspector text fields, and so on—all those acts call setter methods on the inspected object.[5] When a nib file is saved, all the objects with attributes are archived. When the nib file is loaded, they are unarchived by calling initWithCoder.

Since an NSArrayController is an object with attributes, it's archived and unarchived. The same must be true of our PreferencesController, since it has all the same attributes as its superclass.

In fact, our earlier use of init was the exception, not the rule. Because Preferences directly inherits from NSObject, it has no attributes to set with IB, so it doesn't need to be archived; therefore, it can be created without unarchiving, so init is called instead of initWithCoder.

To reassure you that all of this is true, I've created an init (at ❼) that will blow up if it's called.

❷ Setting an NSArrayController's objectClass overrides what IB calls the Class Name attribute. To confirm that, use IB's Attributes inspector to change the PreferencesController's Class Name back to NSMutableDictionary. If setting objectClass didn't override the IB setting, Fenestra would blow up as described in the second step in Section 14.3, *Hooking Up Preference Panel Buttons*, on page 191.

5. Outlet and action connections are the exception: they're described separately from the objects they apply to—which makes sense, since they're properties of pairs of objects, not of individual objects.

❸ and ❹

We need both an NSUserDefaultsController and a data transformer. Creating them is simple. Notice that there's usually only one NSUserDefaultsController per app. Even though you may use one in every nib file, they're all the same object.

❺ Here, at last, is the method that binds. To make it easier to refer to, I'll repeat it here:

Download fenestra/binding-by-hand-2/app/preferences/PreferencesController.rb

```
self.objc_send(:bind, 'contentArray',
               :toObject, @defaults_controller,
               :withKeyPath, 'values.translators',
               :options, {
                 NSHandlesContentAsCompoundValueBindingOption => true,
                 NSValueTransformerBindingOption => @transformer
               })
```

This method call uses an alternative RubyCocoa calling convention. The objc_send method combines its odd-numbered arguments into an Objective-C method name. It then invokes that method, handing it the even-numbered arguments. I think this convention makes calls to methods with many arguments easier to understand.

The method call itself is unexciting. I hope that it's easy for you to see how each argument corresponds to something in the Bindings inspector for an object. In our terminology, the bound object is the PreferencesController itself, the bound name is "contentArray," the rooted keypath is rooted at the NSUserDefaultsController object and extends along the "values.translators" keypath.

Don't look in the API documentation for a class to find its binding names. For example, the documentation for NSArrayController nowhere mentions "contentArray." The best place to find the information is the *Cocoa Bindings Reference* [App08g]. That describes what a binding to a particular name does and which options apply to it. If you just want to know what binding names an object has, you can look at its Bindings inspector in IB and do a mechanical name conversion. (What the inspector calls "Content Array" is, in the code, *"contentArray"*.)

Handling Content as a Compound Value

What does setting a binding's NSHandlesContentAsCompoundValueBindingOption actually do? Here's a situation like that in Fenestra, but it's a bit simpler to explain:

- There are three objects in question: a text field, a TranslatorPreference, and a Controller between them.
- We want the text field to update whenever the TranslatorPreference's display_name property changes. That's accomplished by binding the text field to the controller with the keypath "value.display_name," where value is the name of the controller's property and display_name is one of the properties on TranslatorPreference.
- There's a wrinkle, though: the TranslatorPreference spends most of its time as an NSData. When code changes its display_name, what actually happens is (1) the NSData is unarchived, (2) the display_name is changed, (3) the TranslatorPreference is rearchived, and (4) the new NSData is set as the Controller's value.
- This is a situation somewhat like that on page 187 in that an intermediate key ("value") is given a new object as its value (the new NSData), but we want that signaled as a change to the entire keypath.
- In order to do that, the Controller has to do some work. It has to (1) itself observe the change to value, (2) unarchive the NSData (using the NSValueTransformer given when the binding was made), and then (3) forward on the change notice to any observers of a keypath it roots—just as if no value transformation were involved.

Confusing? A pragmatic way to think about it is this: if you're binding a controller to something and you give a value transformer as a binding option, you have to set NSHandlesContentAsCompoundValueBindingOption too. If you don't, the value transformer will be ignored.

continued

Handling Content as a Compound Value (continued)

However...a *view* will use its value transformer no matter what—NSHandlesContentAsCompoundValueBindingOption is meaningless for views. The lesson? Binding consists of a general-purpose mechanism that's had a lot of assumptions about its use layered onto it. Those assumptions are mainly expressed in predefined options that Apple's NSView and NSController classes understand. For the gory details, see Apple's *Cocoa Bindings Programming Topics* (App08f).

Alternately, you can ask irb:

```
irb(main):001:0> puts NSButton.alloc.init.exposedBindings.to_ruby
target
argument
toolTip
image
...
```

❻ Everything else that you can change in IB's bindings inspector is passed in an options hash. Here, we do the equivalent of selecting the Handles Content as Compound Value checkbox and typing a class name into the inspector's "Value Transformer" text field (except that we provide an already-created instance instead of a class name).

Now you should have a working PreferencesController. Delete the NSUserDefaultsController from the nib file, and test Fenestra.

14.5 What Now?

Most of the preference panel works, but the way it works annoys me. The next part of the book improves the visual behavior of adding and removing rows. Since the final column refers to a Ruby source file, it should be filled using a file chooser dialog box rather than by typing. And since drag and drop has been a feature of the Mac since the very beginning, we'll make it so you can drag and drop a file from the Finder.

Part V

Fun with Tables

Chapter 15

Prologue: Folders and Tests

I discovered an interesting problem when writing the following chapters. Some of them call for more involved changes to Fenestra than you've made so far. But describing those changes unambiguously was *really hard*. I'd write up a checklist for you, set it aside for long enough to forget the details, and then walk through it again—only to find myself making all kinds of frustrating mistakes. I needed a better way to explain to you what was needed.

Enter tests. After explaining some background and giving the rough shape of a solution, I can point to a set of tests, say "Make those tests pass," and be fairly sure you can move smoothly toward working code. You needn't write the tests, just the code that passes them. The code files contain versions with tests that don't yet pass.

The tests will require you to learn about some tools, particularly Shoulda and my own idiosyncratic glue code on top of FlexMock. I'll explain what you need to know as we go.

15.1 Disk Layout

Before we start writing tests, you need to be able to find your way around. As a rule, I put my tests in a folder structure that mirrors that of the source's folder structure. Fenestra's layout is shown in Figure 15.1, on the following page.

The first thing to notice is that I have once again (!) rearranged the app's folder structure. (Annoyed by that? See the sidebar on page 200.) As I was writing tests for the existing code, the old structure kept grating on me. (Tests often increase the annoyance factor of bad decisions. That's one of their virtues.) So, I changed my folder layout to put controllers

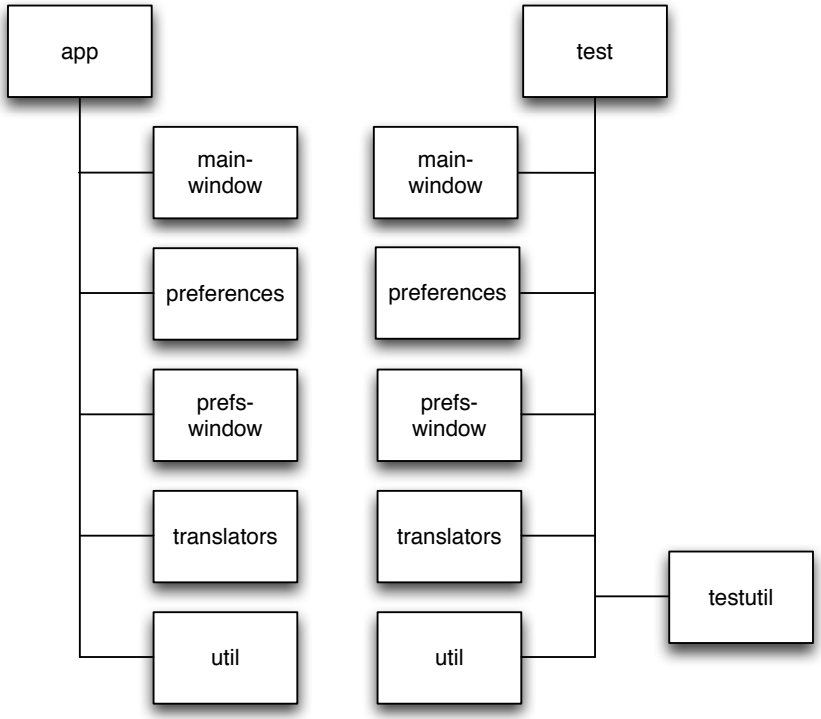


Figure 15.1: A test layout

and other classes specific to one window in a folder named after that window (main-window and prefs-window). Classes or files that live further away from the GUI go elsewhere. If they clump together (all share some responsibility), they go in folders named after that responsibility (preferences and translators). Files that have no better home go into util. They may stay there or, if clumps begin to show up, migrate into other folders.

Moving around files like this would be intolerable if I had to fix up require statements each time. However, the top-level rb_main.rb and app/load-subdirs.rb load all the files, so there's no fiddling with requires.

Many people would separate the controller classes from window-specific classes further from the GUI. The former might go in a controller folder and the latter in a folder often called model. I won't bother with that (at least, not yet).

Fiddling with Structure

When my editor saw that I was once again changing my file system structure, he—being considerate of you, the reader—wrote, "If you are reorging the folder layout, should you backport this structure into previous chapters?" I won't lie to you. My initial reaction was "Rearrange multiple versions of Fenestra? Without tests? Work. . . risk. . . must procrastinate."

While procrastinating, I found myself tinkering with the tests in Chapter 19, *Picking Files with Open Panels*, on page 243. I was renaming, consolidating, making new modules, splitting source into files. . . and I had a crisis of confidence: shouldn't I be *finishing the book*? So, I appealed to my tweeps.* I wrote: "I'd like to watch really expert programmers and see how much time they spend 'fiddling,' moving things around, perfecting structure."

Three better programmers than I wrote back. Corey Haines (coreyhaines) immediately replied, "I would wager that it is a tremendous amount of time," which made me feel better. Brian Button (brianbuttonxp) wrote, "The amount of time spent futzing with the organization or structure of the software is directly proportional to the experience level of that developer," which made me feel positively good. Michael Feathers (mfeathers) wrote, "It might be as eye-opening as (the movie) *The Mystery of Picasso*. Stunning how often he would paint over and radically change," and then Michael quoted a reviewer of the movie as saying "A viewer would be forgiven if, more than once, he felt like screaming at such nonchalant carnage," which convinced me that I wasn't being lazy at all—that I'm leaving changes in to *impart to you a useful software development lesson*.

I urge you to believe that.

*. People who use Twitter, http://www.twitter.com. My Twitter name is marick.

Notice also that the test folder has both util and testutil folders.util contains tests for the app's utilities in app/util. testutil contains utilities used by tests, not tests for any app code.

Running Tests

In the top-level test folder, Rake runs all the tests:

```
$ cd test
$ rake
(in .../fenestra/tdd-4/test)
Started
.............................................
Finished in 2.433668 seconds.
```

Each dot corresponds to a passing test.

In subfolders, Rake runs only the tests in the current folder:

```
$ cd main-window
$ rake
(in .../tdd-4/test/main-window)
Started
.........................
Finished in 2.346468 seconds.
```

While working on a particular test in a subfolder, I'll usually use Rake, since running all the tests takes barely any time longer than running one. But I'll sometimes run just one file:

```
$ ruby app-choice-controller-tests.rb
Loaded suite app-choice-controller-tests
Started
.........
Finished in 0.20482 seconds.
```

Chapter 16

Selections and Editing

In this section, I'll improve the way Fenestra changes the *selection* in the table as you add and delete rows. (You normally select a row by clicking anywhere inside it.) I'll also add code that makes editing new entries more convenient.

16.1 An Example of Creating Tests: The Add Method

When you create a new row in the table, it looks like this:

Display Name	App Name	Favorite	Translator Class
fav	App Name	no	ToString
Display name	App Name	no	ToString

That's lame. The instant after I add a new row, I want to be editing it. That is, I want to see this:

Display Name	App Name	Favorite	Translator Class
fav	App Name	no	ToString
Display name	App Name	no	ToString

I can describe the same behavior in words, not pictures:

```
$ shoulddoc prefs-add-tests.rb
  context "adding a row"
    should "grow the table"
    should "put the new row in the last position"
    should "add a translator preference with default values"
    should "NOT mark the new preference as the favorite"
❶   should "edit initial cell of the new row"
    ...
❷   should "edit first cell in row even if columns have been rearranged"
    ...
```

The advantage of words is that they can easily be attached to executable code—tests—that check whether the app actually implements the desired behavior. Each of the previous should statements is a single test.[1]

The tests before ❶ describe normal NSArrayController behavior. Fenestra shouldn't need any special code to make them pass. I still like to write such tests, both because they help me check whether I really understand the normal behavior and because they prod me into writing the test-support code they require. That way, writing test support code won't get in the way when I'm trying to implement product changes.

The test behind ❶ checks the selection-and-editing behavior described in the two pictures that started the chapter. While I was writing that test, I noticed an ambiguity in my thinking. Did I want the *first* cell in the row selected for editing? Or the Display Name cell? Since Cocoa tables allow the user to rearrange columns, there could be a difference. I decided that I wanted the first cell selected. The reason? When you edit a cell and hit Tab, you begin editing the next cell. Starting in the first cell lets me effortlessly tab through the whole row. The test at ❷ checks whether that decision has been implemented.

One of the benefits of tests is that they help flush out ambiguities that are easy to overlook while coding.

The Structure of a Test File

We'll be working with tests in the file fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb. When you work along with this chapter,

1. shoulddoc is a command I wrote that picks out a test's internal documentation. It's in sandbox/bin. You can also spell it shouldoc.

Download fenestra/table-two-buttons-start/test/prefs-window/sample-tests.rb

```
❶ class StructureSampleTest < Test::Unit::TestCase

❷     def setup
          # ...
      end

      def teardown
          # ...
      end

❸     context "adding a row" do
❹         setup do
              # ...
          end

❺         teardown do
              # ...
          end

❻         should "grow the table" do
              # ...
          end

          should "put the new row in the last position" do
              # ...
          end
      end

      context "removing a row" do
          # ...
      end
  end
```

fenestra/table-two-buttons-start/test/prefs-window/sample-tests.rb

Figure 16.1: The structure of a test file

start from the source in fenestra/table-two-buttons-start. You don't need to write the tests themselves, just the code that passes them.

This test file has a typical structure, shown in Figure 16.1.

❶ A test class inherits from Test::Unit::TestCase, which comes bundled with Ruby. Test classes usually contain many tests. They run automagically when you run the file containing them.

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "grow the table" do
  # ARRANGE
  old_table_length = row_count(@table)

  # ACT
  @add_button.performClick('ignored sender')

  # ASSERT
  assert { old_table_length + 1 == row_count(@sut) }
  assert { old_table_length + 1 == row_count(@table) }
end
```

fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

Figure 16.2: Arrange, act, assert

❷ The setup method runs before each test, and the teardown method runs after it. Most often, it creates objects and instance variables used in all the tests.

❸ "Context" is one of those pleasantly vague words you use when you can't think of the right one or when you're looking for an umbrella term for a bunch of mostly unrelated ideas. In this case, the context serves to group several tests together. Usually the context is a noun phrase that can be used in a sentence starting "The *context* should. . . ." All the tests are about that noun phrase.

In cold practical terms, what a context does is allow you to introduce setup and teardown code that applies to a subset of your tests, as shown at ❹ and ❺. Notice that those are nested method calls (rather than method definitions, as at ❷).

❻ Here, at last, is a test. Its description comes as the argument to should, and its implementation is in the should's block.[2]

The Structure of a Test

My tests usually follow what Bill Wake calls "the three As": arrange, act, assert. You can see an example in Figure 16.2.

2. The context and should methods come from *Shoulda* (http://thoughtbot.com/projects/shoulda/), which is not part of Ruby's standard distribution. I've preinstalled it and another test tool in sandbox.

First the test implementation does test-specific setup, then it makes the code-under-test do something, and finally it evaluates claims about what happened. When writing tests, I usually create those steps in the reverse order:

1. I decide what this test is about: what new behavior do I want the code to have? I write that behavior down as truth statements—*assertions*—that evaluate true when the behavior is correct.
2. Then I decide how a client of the code provokes the new behavior. Perhaps I decide there should be a new method or that an old method should respond differently to certain values passed in. I write down an example of a client acting to provoke the new behavior.
3. Finally, I write any code that needs to run before the action. Most often, that code creates objects referred to in the assert or act code.

The Body of a Test

In this section, I'll explain what's inside the should block in Figure 16.2, on the previous page. I'll explain it in the order I wrote it (assert, act, arrange).

Assert

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "grow the table" do
  # ...

  assert { old_table_length + 1 == row_count(@sut) }
  assert { old_table_length + 1 == row_count(@table) }
end
```

The PreferencesController is supposed to add a TranslatorPreference in response to the clicking of a particular button. In particular, it should do the following:

- Update its internal storage (arrangedObjects) to have one more row than before.
- Do *something* (I don't particularly care what) that causes the table to have one more row than before.

There's more to adding a row than that, but that's all this test checks. Such specificity is a bit silly when you're testing existing behavior.

```
$ ruby prefs-add-tests.rb
  * DEFERRED: adding a row should edit initial cell of the new row.
  * DEFERRED: adding a row should edit first cell in row even if ↩
              columns have been rearranged.
Loaded suite prefs-add-tests
Started
..F.
Finished in 0.501686 seconds.

  1) Failure:
test: adding a row should grow the table. (PrefsControllerActionTest)
    [.../assert2-0.3.2/lib/assert2.rb:221:in `_flunk_2_0'
     .../assert2-0.3.2/lib/assert2.rb:75:in `assert_'
     .../assert2-0.3.2/lib/assert2.rb:54:in `assert'
     prefs-add-tests.rb:75:in `__bind_1228515269_448656'
     .../Shoulda-1.1.1/lib/Shoulda.rb:189:in `call'
     .../Shoulda-1.1.1/lib/Shoulda.rb:189:in `test: adding a row should ↩
grow the table. ']:
assert{ ( old_table_length + 2 ) == row_count(@table) } --> false
        old_table_length     --> 5
    ( old_table_length + 2 ) --> 7
             @table          --> #<...>
       row_count(@table)     --> 6.
```

Figure 16.3: A verbosely failing test

When you're creating new behavior, though, it's valuable. By writing narrow tests, you can make them pass one at a time. That gives you constant feedback that you're heading down a reasonable path, and it makes debugging trivial—if you expected the test to pass and it still fails, something you *just did* must be wrong.

To write assertions, I use Phlip's *Assert{2.0}*.[3] If either of the assert blocks evaluates false, the test fails.

Let's look at a failure. Change one of the 1s in the assertion to a 2. You should see something like Figure 16.3.

Ignore the messages about deferred tests for now. The stack dump contains a number of methods I don't care about, so I've written a command tst (also in the sandbox) that filters them out.

3. http://assert2.rubyforge.org/

```
$ tst prefs-add-tests.rb
...
  1) Failure:
test: adding a row should grow the table. (PrefsControllerActionTest)
     prefs-add-tests.rb:75:in `__bind_1228515595_378452'
assert{ ( old_table_length + 2 ) == row_count(@table) } --> false
        old_table_length     --> 5
    ( old_table_length + 2 ) --> 7
            @table           --> #<...>
       row_count(@table)     --> 6.
```

Notice how kind assert is. Not only does it alert us to the failure, but it shows how each subexpression evaluated.[4] If you're running the test in Terminal or iTerm (as opposed to within an editor window), the output is even color-coded.

The only line of the stack dump gives the line number of the failing assert. If your editor is kind, it will give you an easy way to jump to the failing statement.[5]

Let's return to the *assert* part of the test:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "grow the table" do
  # ...

  assert { old_table_length + 1 == row_count(@sut) }
  assert { old_table_length + 1 == row_count(@table) }
end
```

Notice there are two instance variables: @table and @sut. My convention is that a class I'm testing is represented by an instance named @sut. That's tester jargon that means "system under test." I could have named it something like @preferences_controller, but @sut makes its distinct role more obvious.

Throughout the test, the @sut is surrounded by other objects: there's an NSTableView, two NSButton objects, an NSUserDefaultsController, and so on. The setup method creates them, connects them together, and gives them names like @table and @add_button. All the different tests for the

4. It does this by reevaluating each subexpression. If any of them have side effects such as setting instance variables, those side effects will be run more times than if the test succeeded. That could be confusing—"Why is @counter equal to 2 instead of 1?"—but it's pretty odd to want assertions to have side effects.
5. If you use Emacs, look in sandbox/elisp/ruby-visit-source.el for some Elisp code that adds that feature.

PreferencesController use the same names. (You'll see more about setup methods in Section 16.4, *Building Setup Methods*, on page 227.)

Finally, notice the method row_count. A rule of thumb I use for tests is that they should use roughly the words you'd use if you were describing some code behavior to a knowledgeable person over the phone. If you'd skip a detail on the phone, hide it behind a helper method.

Using a helper method also lets you hide inconvenient facts. It happens that you get one count of rows with code like this:

```
@sut.arrangedObjects.count
```

. . . and the other with code like this:

```
@table.numberOfRows
```

. . . but I really don't want every last test rubbing those details in my face. So, I use this instead:

Download fenestra/table-two-buttons-start/test/prefs-window/testutil/table-support.rb

```
def row_count(thing)
  if thing.is_a? NSArrayController
    thing.arrangedObjects.count
  else
    thing.numberOfRows
  end
end
```

That's it for the assert step, which is all about desired results. Now let's make the @sut produce them.

act

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "grow the table" do
  # ...
  @add_button.performClick('ignored sender')

  assert { old_table_length + 1 == row_count(@sut) }
  assert { old_table_length + 1 == row_count(@table) }
end
```

We are testing the behavior of PreferencesController's add method (the action connected to the @add_button). I could call that method directly:

```
@sut.add('ignored sender')
```

Instead, I've chosen to call add indirectly, through the NSButton's performClick method. Many people would point out that I'm testing two

things—whether the button is hooked up correctly and what add does—and that when the test fails, I won't know which I got wrong. They're absolutely right. However, I've noticed that a lot of my RubyCocoa flailing around is because of my misunderstanding of how some Cocoa control really works. Because of that, I use real controls in my tests whenever I can, in the hopes of stumbling over misunderstandings as early as possible.

Arrange

What has my test used that isn't set up yet? Since the setup creates the instance variables, there's only one variable left over: old_table_length.

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "grow the table" do
  old_table_length = row_count(@table)

  @add_button.performClick('ignored sender')

  assert { old_table_length + 1 == row_count(@sut) }
  assert { old_table_length + 1 == row_count(@table) }
end
```

What Do Programmer Tests Test?

Consider this code:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "add a defaulted translator preference" do
  @add_button.performClick('ignored sender')

  names = @sut.valueForKeyPath('arrangedObjects.display_name')
❶ assert { names.last == DEFAULT_TRANSLATOR_DISPLAY_NAME }
end
```

The assertion at ❶ checks the value of only one TranslatorPreference field. Why not the others?

As J. B. Rainsberger has said, "You stop testing when fear turns into boredom." The purpose of testing is to produce *justified confidence* that the code does as you intended. Given what I know, checking one column suffices.

Notice also that the assertion at ❶ checks the actual value against a symbolic constant. That constant is defined within Fenestra, here:

Download fenestra/table-two-buttons-start/app/util/Constants.rb

```
DEFAULT_TRANSLATOR_DISPLAY_NAME='Display name'
DEFAULT_TRANSLATOR_APP_NAME='App Name'
DEFAULT_TRANSLATOR_CLASS_NAME='ToString'
```

The test is *not* checking that the default value is spelled correctly, capitalized appropriately, or otherwise sensible. The best way to check that is to click the Add button and look at a new row. Better yet would be to have *someone else* do the looking and then tell me whether what she sees helps her understand what to do next.

A Downside of Using Controls

This test checks whether the row is put in the right place:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "put the new row in the last position" do
❶   old_names = @sut.valueForKeyPath('arrangedObjects.display_name')

    @add_button.performClick('ignored sender')

    new_names = @sut.valueForKeyPath('arrangedObjects.display_name')
❷   assert { old_names == new_names[0...-1] }
end
```

(See Section 14.2, *Using Rooted Keypaths in Code*, on page 189 if you don't understand how line ❶ works.)

Notice that the assertion at ❷ checks the contents of only the @sut, not the @table. Why not? After all, the test in Figure 16.2, on page 205, checked both.

I couldn't check the table because there's no NSTableView method that lets me ask the table about what it thinks it should display. Real controls are often annoying that way because testing used to be something programmers didn't do, so they didn't need to query controls, so framework builders spent their time on other things, things programmers wanted. As more programmers start using tests, that's starting to change.

I have confidence, though, that I don't need the check—after all, I know from the test in Figure 16.2, on page 205, that the table adds a new row, and it seems hard to believe that it could be at a different index

than its equivalent in arrangedObjects. Also, I always try a new feature after I add it, and it's hard to believe I'd miss such an obvious problem.

You'll see a case where I made a different decision in Section 16.2, *Working with an Uncooperative Control*.

One More Wafer-Thin Fact

assert fails the test if the block's value isn't true. deny, shown next, fails the test unless the value *isn't* true.[6]

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "NOT mark the new preference as the favorite" do
  @add_button.performClick('ignored sender')

❶   deny { @sut.arrangedObjects.last.favorite }
end
```

16.2 Working with an Uncooperative Control

There are two kinds of tests inside prefs-add-tests.rb: ones introduced with should and ones introduced with should_eventually. At this point, the ones using should test plain old NSArrayController behavior. Here's what happens when you run them:

```
$ cd fenestra/table-two-buttons-start/test/prefs-window
$ tst prefs-add-tests.rb
  * DEFERRED: adding a row should edit initial cell of the new row.
  * DEFERRED: adding a row should select all text in edited cell.
  * DEFERRED: adding a row should edit first cell in row even if ↩
              columns have been rearranged.
  * DEFERRED: adding a row should select the rest of the row so that ↩
              tabbing out of first cell moves to the second.
Loaded suite prefs-add-tests
Started
....
Finished in 0.546346 seconds.

4 tests, 0 assertions, 0 failures, 0 errors
```

6. In Section 16.1, *What Do Programmer Tests Test?*, on page 210, I said I need to check only one field of the new row to get confidence it'd been filled with the default values. Why, then, have I gone out of my way to check the value of the favorite field in this test? I had an internal design debate about whether adding a new row should make it the favorite. I decided against it, and the test documents that decision.

Yawn. PreferencesController does what we already knew it did. The interesting tests—tests for the behavior it should add to NSArrayController—don't run.

So, now it's time to work on the new behavior. Let's make the first deferred test pass. Here, with some hand waving, is how the assert part of it should look:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should "edit initial cell of the new row" do
  # ...
  assert { ... first cell of last row is edited ... }
end
```

How can we know what table cell, if any, is being edited? As far as I know, we can't. NSTable doesn't let you ask that question. I guess we have to give up.

Just kidding! If we can't check that the correct behavior is caused, we can at least check that the behavior is caused correctly: does the PreferencesController send the right message or messages to its NSTable?

If you look at the NSTable API documentation, you'll find that the editColumn_row_withEvent_select is the message to send. How can we tell whether it's sent correctly? Like this:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should_eventually "edit initial cell of the new row" do
  new_row_index = row_count(@table)
  during {
    @add_button.performClick('ignored sender')
  }.behold! {
    @table.should_receive(:editColumn_row_withEvent_select, 4).once.
           with(0, new_row_index, any, any)
  }
end
```

You can think of this code as replacing—for the duration of the test—the @table's editColumn_row_withEvent_select method with a new version that records how it's called.[7] If performClick or the methods it uses never call editColumn_row_withEvent_select—or if it's called twice—the test will fail. It will also fail if the column index is anything other than 0 or the row index is anything other than the location of the newly added row.

7. In reality, the method is replaced on a newly created subclass of NSTableView. Otherwise, Objective-C would have a temper tantrum. You'll see how "programmable" objects like @table are created in Section 20.5, *Warm-up: Pathname Methods*, on page 269.

The test, however, doesn't care about the other two method arguments; they can be *any*thing.

But that's not a full description of how editColumn_row_withEvent_select should be called. The last argument passed to it is a boolean that determines whether the existing text in the cell is selected. Selecting it sounds good—that way, a single keypress erases "Display Name." I could make our current test care about that by changing the second any to true. That, however, leaves an important decision somewhat obscure, so I'll highlight it in another test:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should_eventually "select all text in edited cell" do
  during {
    @add_button.performClick('ignored sender')
  }.behold! {
       @table.should_receive(:editColumn_row_withEvent_select, 4).once.
              with(any, any, any, true)
  }
end
```

About Debugging

Moving back to the first deferred test, I want to make it pass. That seems easy enough:

Download fenestra/table-two-buttons-start/app/prefs-window/PreferencesController.rb

```
def add(sender)
  super_add(sender)
  @table.objc_send(:editColumn, 0,
                   :row, arrangedObjects.size-1,
                   :withEvent, nil,
                   :select, true)
end
```

Or maybe not:[8]

```
  1) Failure:
test: adding a row should edit initial cell of the new row.
     .../app/prefs-window/PreferencesController.rb:53:in `add'
     prefs-add-tests.rb:134:in `performClick'
     prefs-add-tests.rb:134:in `__bind_1228936193_588633'
     prefs-add-tests.rb:133:in `__bind_1228936193_588633'
in mock 'table': ↩
no matching handler found for editColumn_row_withEvent_select(0, 4, nil, 0)
```

8. If you want to run the test along with me, you'll have to convert should_eventually to a should.

I wrote an *expectation* that the new row index would be 5 (the index of the last row in the updated table), but it's actually 4. The expectation is not met. How could that be? I added some debugging text:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
      should_eventually "edit initial cell of the new row" do
        new_row_index = row_count(@table)
❶       puts "expected row index = #{new_row_index}"
        during {
          @add_button.performClick('ignored sender')
❷         puts "table count: #{row_count(@table)}"
❸         puts "sut count: #{row_count(@sut)}"
        }.behold! {
#         @table.should_receive(:editColumn_row_withEvent_select, 4).once.
#                with(0, new_row_index, any, any)
        }
      end
```

That showed this:

```
..expected row index = 5
table count: 6
sut count: 6
```

The row counts have the right values *after* the controller's add method is called but somehow don't *during* the call. Can that be true? A simple puts in add will show us:

Download fenestra/table-two-buttons-start/app/prefs-window/PreferencesController.rb

```
def add(sender)
  super_add(sender)
❶ puts "after super_add #{arrangedObjects.size}"
  @table.objc_send(:editColumn, 0,
                   :row, arrangedObjects.size-1,
                   :withEvent, nil,
                   :select, true)
end
```

And. . . ?

```
.expected row index = 5
after super_add 5
table count: 6
sut count: 6
```

The superclass's add is doing something funny. It's time to look in the documentation!

add:

Creates and adds a new object to the receiver's content and arranged objects.

- (void)add:(id)*sender*

Parameters

sender

Typically the object that invoked this method.

Special Considerations

Beginning with Mac OS X v10.4 the result of this method is deferred until the next iteration of the runloop so that the error presentation mechanism can provide feedback as a sheet.

We don't have a run loop in the test, but there's apparently something equivalent going on. A little poking in the API documentation shows a better method:

Download fenestra/table-two-buttons-start/app/prefs-window/PreferencesController.rb

```
def add(sender)
❶  addObject(TranslatorPreference.alloc.init)
  @table.objc_send(:editColumn, 0,
                   :row, arrangedObjects.size-1,
                   :withEvent, nil,
                   :select, true)
end
```

That passes the current test.

My Point, and I Do Have One

Like all debugging, this debugging-via-tests is painful, but doing the same debugging by starting Fenestra after each change would be more painful.

More Debugging

When I added editColumn_row_withEvent_select to PreferencesController, I had to pick some value for the fourth argument even though the test made no demands on it. True to my wildly proactive nature, I picked true: exactly what the next test expected.

So, that test ought to pass easily:

```
  1) Failure:
test: adding a row should select all text in edited cell.
     prefs-add-tests.rb:26:in `teardown']:
in mock 'table': method 'editColumn_row_withEvent_select↩
       (ANY, ANY, ANY, true)' called incorrect number of times.
<1> expected but was
<0>.
```

Hmm. I see two possibilities:

- editColumn_row_withEvent_select is never called at all.
- It's called, but its fourth argument is wrong. (It can't be one of the first three, since any value is OK.)

To decide between the two, I'll print the fourth argument:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
# A version with debugging info printed.
@table.should_receive(:editColumn_row_withEvent_select, 4).once.
       with(any, any, any,
            on { | arg |
              puts "select text: #{arg.inspect}"
              arg == true
            })
```

The on construct takes the value given in its position and passes that value to its block. If the block's return value is true (that is, not nil or false), the argument value matches what was expected. But the block can do whatever it wants before returning a value—like print debugging information.

Here's the result:

```
% tst prefs-add-tests.rb
...
Started
.....edit column val: 1
```

Ah. The true variable sent down into the Cocoa universe as the last argument somehow loses its "booleanness" and comes out as an integer. So, the test has to be changed to replace this:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
@table.should_receive(:editColumn_row_withEvent_select, 4).once.
       with(any, any, any, true)
```

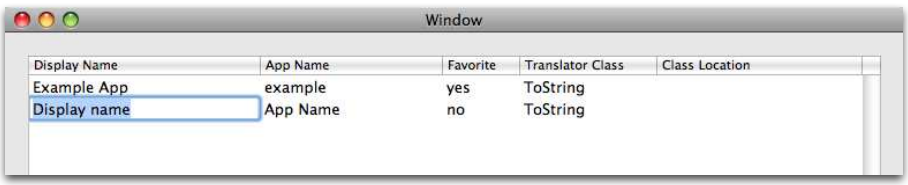
with this:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

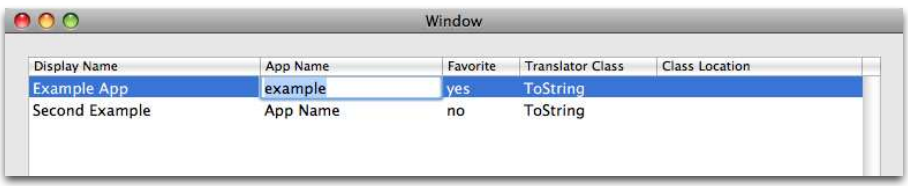
```
@table.should_receive(:editColumn_row_withEvent_select, 4).once.
       with(any, any, any, 1)
```

Trying the Code

Time to try out the app! Start Fenestra, open the preference pane, click the Add button, and you should see this:

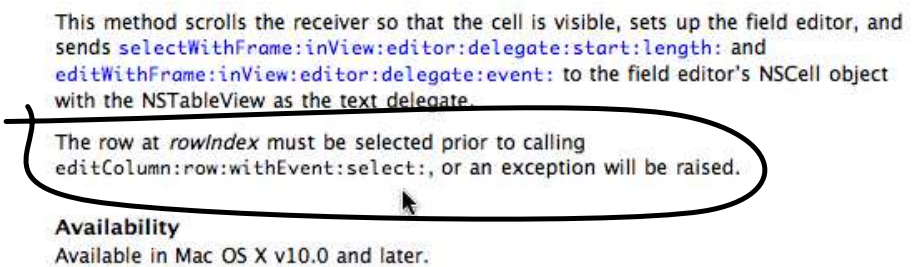


Success! But now change the display name and hit Tab. Oops:



Tabbing goes to a different row, the first one. Why? Here's a clue: continuing to tab moves you from column to column in the first row. Another clue is that the first row is highlighted in the "after" picture, but the second line isn't in "before." Perhaps selecting (as with a single-click) the newly added row would help?

Indeed, it should, because the API documentation for editColumn_row_withEvent_select says this:


This method scrolls the receiver so that the cell is visible, sets up the field editor, and sends selectWithFrame:inView:editor:delegate:start:length: and editWithFrame:inView:editor:delegate:event: to the field editor's NSCell object with the NSTableView as the text delegate.

The row at *rowIndex* must be selected prior to calling editColumn:row:withEvent:select:, or an exception will be raised.

Availability

Available in Mac OS X v10.0 and later.

We don't get the promised exception, but we do get bad behavior. Before fixing the problem, I'd like to write a test that talks explicitly about

what behavior results from a Tab after adding a row. That is, I want a test that has the word *tab* in it. Alas, Cocoa gives me no programmatic access to tabbing (as far as I know), so I instead have to write a test about selections and assume that a selected row implies the right tabbing behavior.

Here's the test:

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
should_eventually "select the rest of the row so that tabbing out " +
                  "of the first cell moves to the second" do
  @add_button.performClick('ignored sender')

  assert { selection_indexes(@sut) == [row_count(@sut) - 1] }
  assert { selection_indexes(@table) == [row_count(@table) - 1] }
end
```

And here's the code that passes it:

Download fenestra/table-two-buttons-start/app/prefs-window/PreferencesController.rb

```
def add(sender)
  addObject(TranslatorPreference.alloc.init)
  selection = NSIndexSet.indexSetWithIndex(arrangedObjects.size-1)

  @table.objc_send(:selectRowIndexes, selection,
                   :byExtendingSelection, false)
  @table.objc_send(:editColumn, 0,
                   :row, arrangedObjects.size-1,
                   :withEvent, nil,
                   :select, true)
end
```

Are you going to believe that passing the test means tabbing works? I hope not. I hope you try it.

My Point, and I Do Have One (Two, Actually)

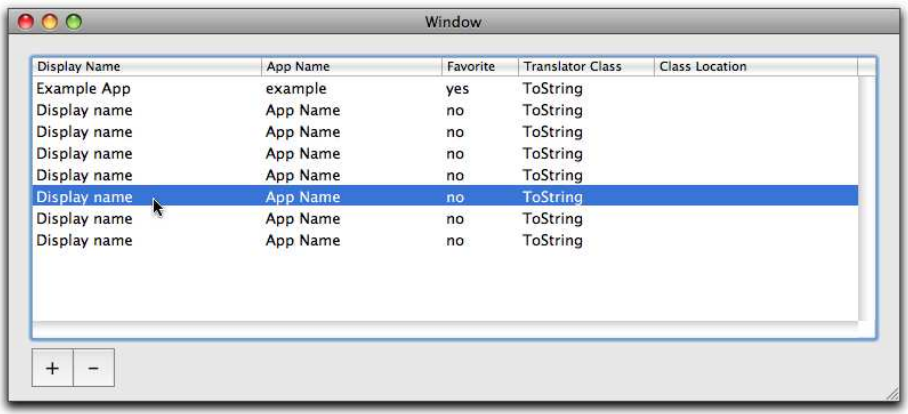
- Tests make development of a UI change less of a hassle because you do less running of the app, but they do not *replace* running the app. Always check a feature after you think you've finished it.
- But do more than merely glance admiringly at your finished work. If I'd just looked at the result of pressing the Add button, I might not have noticed or cared that the row wasn't selected. It was only by emulating what a real user would do that I stumbled across a problem a real user would find.

16.3 Try This Yourself

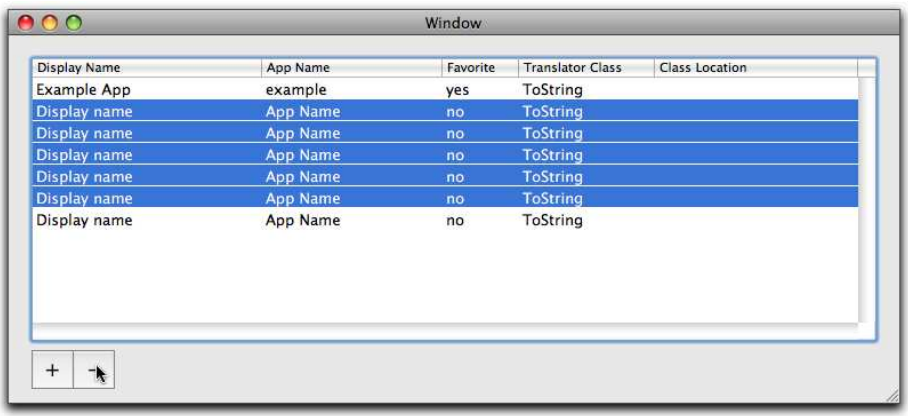
With what you know, you can make two more improvements to the preference panel.

Make Row Removal More Convenient

Removing a row has always been peculiar. Working with earlier versions of Fenestra, I noticed that removing a selected row would always leave the first row selected. That was *never* the next row I wanted to delete. In this version, it's gotten even stranger. In the following picture, I'm about to delete the selected row:



In the next one, I have this:



What's with all the selected rows? Seeing them makes me realize two things:

- It sure would be nice to have some mechanism. . . some script, or group of scripts, that could run frequently. . . to tell me when app behavior has changed.
- I need to think through how I want row deletion to behave and write code that adds that behavior.

Tests are my preferred tool for both goals. And here they are:

```
context "removing a row"
  should "remove the currently selected row from the table"
  should "remove the currently selected row from the preferences"
  should "do nothing if no row is selected"
  should "remove multiple preferences if multiple rows selected"
  should_eventually "select the next row, if there is one"
  should_eventually "even select the next row if given a selected range"
  should_eventually "even select next row if given a disjoint set"
  should_eventually "select the last table row, if there is no next row"
  should_eventually "even select the last row if given a contiguous range"
  should_eventually "even select the last row in a disjoint range"
  should_eventually "select nothing if the table is now empty"
```

The undeferred tests check normal NSArrayController behavior, and they all pass now. Your job is to make the rest of them pass.

You can find my solution in fenestra/table-two-button-end/app/prefs-window/PreferencesController.rb.

Hints

- Like add, NSArrayController's remove method doesn't change the arrangedObjects array until the next iteration of the run loop. You can actually take advantage of that to write inscrutable code that looks like it shouldn't work but does. Let your conscience be your guide.
- NSArrayController has no fewer than seven methods whose names begin with remove. Learn from my mistake: if the one you've picked isn't working, don't beat your head against the wall for too long before trying another one.
- If you want to see what rows are in arrangedObjects, you may find this:

  ```
  puts valueForKeyPath('arrangedObjects.display_name')
  ```

  . . . less annoyingly verbose than this:

  ```
  puts arrangedObjects
  ```

- NSIndexSet objects can be converted to Ruby arrays with to_a, so you can easily use both Array and NSIndexSet methods in your solution.
- Don't forget to use Fenestra before considering yourself done.

Interdependent Favorite Values

Here, again, is a preference pane:

Observe with horror the result of changing a favorite:

Making a new favorite should unmark the old favorite. Your task is to fix the problem *by changing PreferencesController* to make these tests pass:

```
$ shouldoc prefs-consistency-tests.rb
    context "choosing a favorite"
      should_eventually "cause any old favorite to cease being favorite"
      should_eventually "propagate change into the defaults controller"
      should_eventually "not be fooled by two identical rows"
    context "making the favorite not favorite"
      should "cause all preferences to be not-favorite"
    context "resetting a non-favorite to non-favorite"
      should "leave the favorite alone"
    context "a change to any other keypath"
      should "do nothing"
```

I emphasize changing PreferencesController because that is really the wrong place to do the work. The rule that there can be only one favorite is a *business* rule, a rule about the domain in which we're working. It should be enforced further from the user interface—below the controllers in what's often called either the *business logic* or the *model*. However, doing that wouldn't teach you as much about Cocoa.

Implementation Background

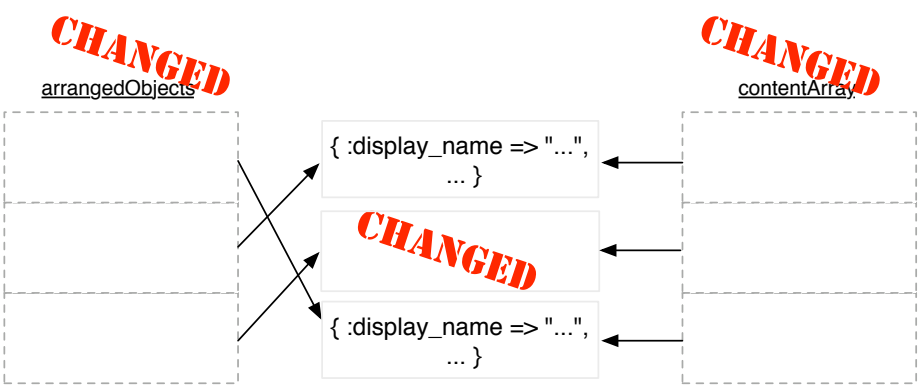
To date, our PreferencesController code hasn't had to concern itself with what happens when a user finishes editing a cell. Somehow, mysteriously, its superclass has been accepting edits, changing the arrangedObjects array, and managing to update the NSUserDefaultsController too. Now we have to—somehow—participate in that process by making an edit of one cell update another. A bit of the mystery has to be revealed.

When editing is finished, an entry in the arrangedObjects array is updated. In a test, that would look like this:

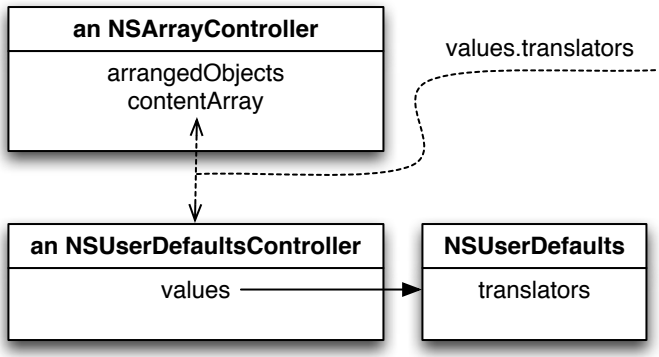
Download fenestra/table-two-buttons-start/test/prefs-window/prefs-consistency-tests.rb

```
@sut.arrangedObjects[index].favorite = value
```

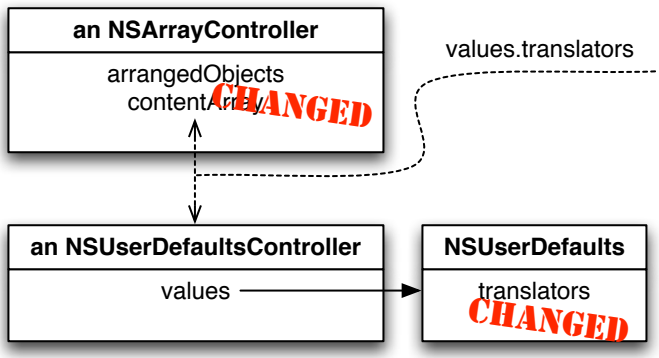
But that causes a cascade of events, all of which are handled by code inside the NSArrayController superclass. First, a change to one of the arrangedObjects is the same thing as a change to the content array:



But the content array is bound to the user defaults, via an NSUserDefaultsController:



That means it's the job of the NSArrayController to somehow notice the changed content array and update the value at the end of the keypath "values.translators":



The "somehow" of "somehow notice" is because of the infrastructure of bindings, which you can learn about in Chapter 27, *The Underpinnings of Cocoa Bindings*, on page 350. The operational definition of "notice" is that a particular NSArrayController method is called. If you were to call it in a test, the call would look like this:[9]

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-consistency-tests.rb

```
@sut.objc_send(:observeValueForKeyPath, 'favorite',
               :ofObject, @sut.arrangedObjects[index],
               :change, nil,
               :context, nil)
```

9. In the real app, the final two arguments aren't nil, but you won't need them for this chapter. I'll leave their explanation to Section 27.4, *Observing Changes*, on page 353.

That method call is the place where you can intervene to make a finished edit do more than affect one cell. To improve Fenestra, you'll override observeValueForKeyPath_ofObject_change_context to add a little behavior.

The Tests

A typical test proceeds something like this:

1. Each test begins with four TranslatorPreferences, the first one being the favorite:

   Download fenestra/table-two-buttons-start/test/prefs-window/prefs-consistency-tests.rb

   ```
   prefs = [{ :display_name => 'original bff', :favorite => true },
            { :display_name => 'new bff', :favorite => false },
            { :display_name => 'identical', :favorite => false },
            { :display_name => 'identical', :favorite => false }
           ]
   make_fake_defaults_controller(*prefs)
   ```

2. The act part of the test marks one of the TranslatorPreferences as either the favorite or not favorite, referring to it by index:

   Download fenestra/table-two-buttons-start/test/prefs-window/prefs-consistency-tests.rb

   ```
   assert { favorites(@sut) == [true, false, false, false] }
   make_this_the_favorite(1)           #^
   ```

   (I use assertions before the action as a bit of documentation of the starting arrangement. That makes the tests easier to read.)

3. Finally, the test asserts something about the "favoriteness" of each TranslatorPreference:

   Download fenestra/table-two-buttons-start/test/prefs-window/prefs-consistency-tests.rb

   ```
   assert { favorites(@sut) == [false, true, false, false] }
   ```

You can find my solution in fenestra/table-two-button-end/app/prefs-window/PreferencesController.rb.

Before running Fenestra again, delete your existing user defaults:

```
% rm ~/Library/Preferences/com.apple.rubycocoa.FenestraApp.plist
```

You should do that because I've changed the format of the TranslatorPreference to include a permanent identity. That's explained in the following hints.

Hints

- Do not forget to call the superclass method in your overriding method.
- Changing one TranslatorPreference's favorite to true may mean changing another one's to false. The following will *not*, by itself, do what you want:

  ```
  arrangedObjects[old_favorite].favorite = false
  ```

  You'll find that the NSArrayController doesn't "notice" this change, so it doesn't send it down the rooted keypath to the user defaults. The changes it will notice are the addition or removal of entire TranslatorPreferences. See removeObjectAtArrangedObjectIndex and similar methods.
- Checking whether two arrangedObjects elements are the "same" is tricky:
  - Equality (==) is defined for TranslatorPreferences, but two rows in the table can easily have the same contents. So if you search for objects using equality, you might find the wrong one. There's a test to protect against that mistake.
  - Object identity (object_id) isn't stable. If you add *any* object to the arrangedObjects, *all* of them are reloaded—which changes their object_ids. The same happens if you delete any element. So if you set a variable to a preference, update the array, and then use that variable in a search, you'll get bad results. *There is no test to prevent this bug.*[10]

  I've added a globally unique identifier to TranslatorPreference. You can tell whether two objects "mean the same thing"—at one point started out as a single object—like this:

  Download fenestra/table-two-buttons-end/app/prefs-window/PreferencesController.rb

  ```
  next if pref.was_originally_identically?(new_favorite)
  ```
- Try your solution out when the tests pass. You will probably notice that tabbing out of the Favorite column doesn't work right. I tried to make it work, but it turns out to be surprisingly difficult. That

10. I couldn't see a way to write one that didn't make you pay more attention to the ins and outs of testing than I expect you want to at this point.

made it even clearer that I should be solving the problem in a different object—in the business logic instead of GUI logic—so giving up was an easy decision. I suggest you give up too or solve it outside of PreferencesController.

Extra Credit

Table columns are sortable, but none of the tests or code takes account of that. As a result, there are at least two bugs associated with adding and removing. Find and fix them.

16.4 Building Setup Methods

If we've both been lucky, the test code was clear enough that you never had to care what the setup methods did. Nevertheless, you may be curious about them. (Skip this section if you're not.)

The way I create setup methods is a straightforward microcosm of a lot of on-the-fly design:

1. I write some code in the arrange part of a test.
2. Working on a later test, I find I need the same code. I promote it to the closest enclosing setup method.
3. As the setup code gets more complicated, I break it down into smaller methods with better names. Those methods migrate into utility files when they're used in more than one test file.
4. As I keep reusing the methods, I keep changing their names and trying to fit them into some sort of mental structure that makes them easier to remember.
5. Eventually, the "setup framework" stabilizes into something I can use comfortably.

In the Fenestra tests, the setup framework's naming is inspired by the question parents ask of spoiled children: "Do you think the whole universe revolves around you?" In the case of the system under test, the answer is yes: the whole universe—of the tests—*does* revolve around it. So, the tests for the Add button are set up as shown in Figure 16.4, on the following page.

Download fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

```
def setup
  super
❶ prefs = [{ :display_name => 'originally at 0' },
           { :display_name => 'originally at 1'},
           { :display_name => 'originally at 2' },
           { :display_name => 'originally at 3' },
           { :display_name => 'originally at 4' }]
  make_fake_defaults_controller(*prefs)

❷ @sut = @preferences_controller = PreferencesController.alloc.init
❸ the_universe_revolves_around(@sut)
❹ connect_objects_in_universe
❺ awaken_all_objects
end

def teardown
❻ disconnect_objects_in_universe
❼ super
end
```

fenestra/table-two-buttons-start/test/prefs-window/prefs-add-tests.rb

Figure 16.4: An idiom for setup and teardown

❶ Most objects are used the same way in every test, so they're not mentioned specially. These particular tests check how the PreferencesController interacts with Cocoa defaults. Since using the real defaults system is fragile (what if I run two tests simultaneously?), I make a fake defaults object at ❶.

❷ I always initialize the @sut explicitly, even though it could be done in a helper method, to reinforce what class these tests are about.

❸, ❹, ❺ I separate the next three steps so that it's clearer that setup mimics loading a nib file. At ❸, all the objects are created. (In a real nib, that's done mainly by calls to initWithCoder.) Then, at ❹, outlets are connected. Finally, at ❺, each object's awakeFromNib method is called. In real life, objects would be ready to receive method calls from the run loop; in tests, the methods are called from test code.

My setup code is a bit unusual because instance variables are set behind the scenes. I rely on naming conventions and habit to know their names.

The teardown code cleans up after the test.

❻ Outlet connections don't require explicit disconnection, but notifications do. If a set of tests uses notification observers (these do not), failure to disconnect in teardown means objects from old tests will receive notifications sent by new ones, which can be damnably confusing.

❼ It happens that the Test::Unit::TestCase class's normal teardown method does nothing, yet it's nevertheless vitally important you call it. The reason is that the FlexMock code behind should_receive methods (like the one you saw in Section 16.2, *Working with an Uncooperative Control*, on page 212) installs its own teardown method in Test::Unit::TestCase. That version of teardown contains the code that actually checks whether all the messages that should have been received have been. If you leave off the super call, that code will never run, and the test will *appear to pass*. And that would be bad. It's a good habit to run a new test and watch it fail before adding the code to make it pass. Nevertheless, you may be curious about them.

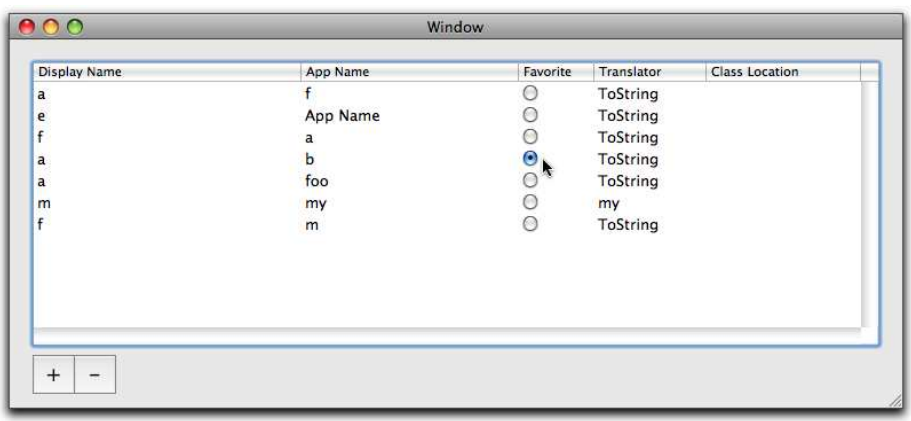
16.5 What Now?

That was somewhat involved. Let's cleanse our palate by picking a task that can be done almost entirely in Interface Builder: adding radio buttons to the table.

Chapter 17

Buttons in Tables

Users don't want to type "yes" or "no" into the Favorite column. It acts like a group of radio buttons would: everything in one of two states, and only one at a time in the "on" state. *And* we had to do extra work to provide this inconvenience: we had to write code to make sure they type the right words. Feh. What we want is this:



A tower of radio buttons requires just one substitution in Interface Builder. But first, I'll give you a smidge of background.

17.1 Cells

In the book so far, controls (subclasses of NSControl) have taken credit for displaying values and responding to events. Behind the scenes, though, something else has been doing the real work. (A disjunction between work and credit. . . I wonder where I've seen *that* before.) The worker is a *cell*, an object that inherits from NSCell. The advantage of

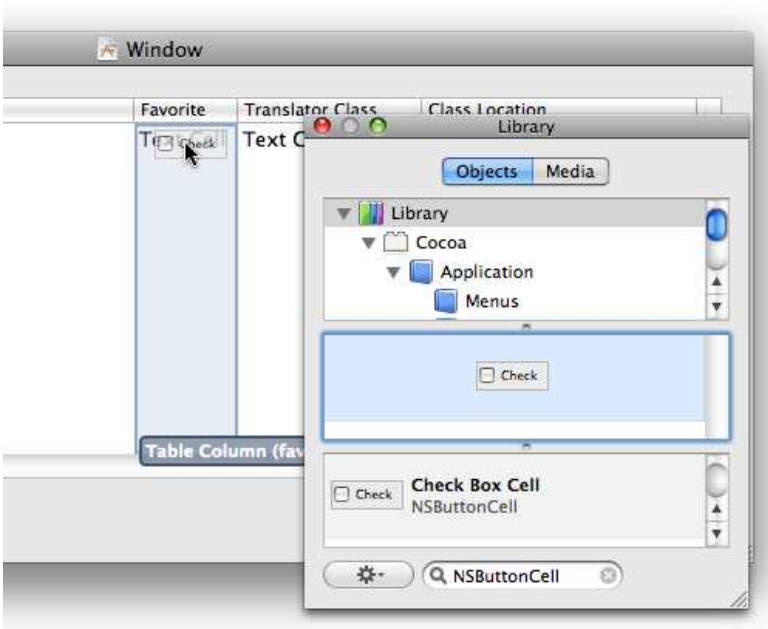
the separation is that it makes it easy for a control (a table column, say) to be paired with different classes of cell objects depending on whether you want to display text or colors or images or. . . buttons.

The NSTableColumn for the Favorite field works with a single NSTextFieldCell to display and edit text that represents boolean values. Let's use an NSButtonCell instead.

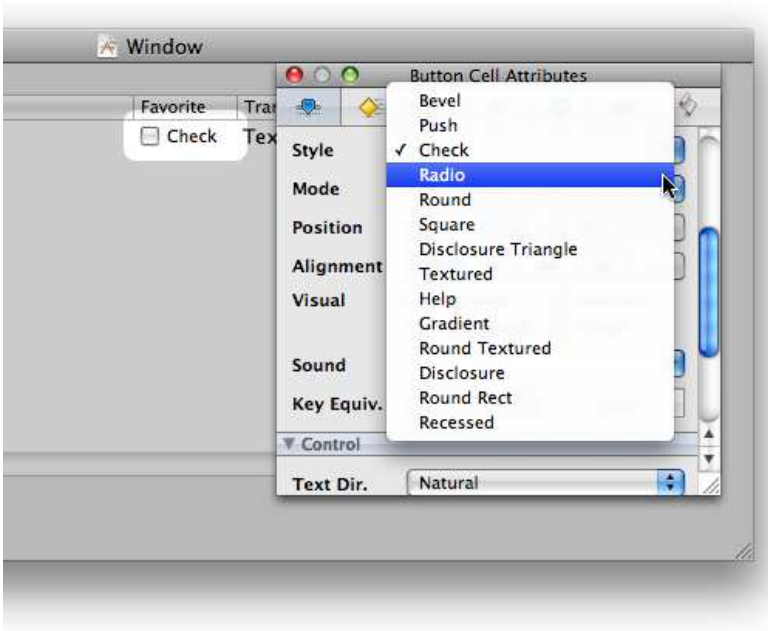
17.2 Making the Change

You can change the code in the version of Fenestra you finished in the previous chapter, or you can use the one in fenestra/table-two-buttons-end. To see how I made the change, look at fenestra/table-radio-buttons.

- Interface Builder's library contains only one kind of NSButtonCell: one meant to be used in checkboxes. You'll fix that shortly; for now, just drag it onto the table column and drop it:

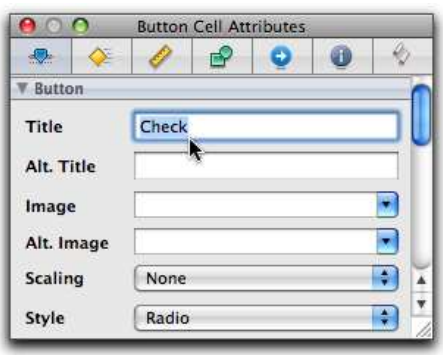


- Use the Attributes inspector to change the cell's style to Radio:



At this point, my version of Interface Builder tries to help me by squeezing all the table columns to their minimum widths. I hope yours is less helpful. If not, drag the edges of the header cells to make the columns wider.

- In the same inspector, clear the Check value since it doesn't make sense for any label to be repeated in each column cell.



- Remove the BooleanCellFormatter from the doc window, since it's no longer needed.

- Fenestra should work fine now. You might want to tidy up by removing app/prefs-window/formatters.rb, which has the now-abandoned BooleanCellFormatter.

That's all there is to it.

17.3 What Now?

Table columns are a bit unusual in that we needed specific code to implement the one-button-at-a-time-is-on rule. More commonly, a set of radio buttons is wrapped inside an NSMatrix that implements the rule. See Apple's *Button Programming Topics for Cocoa* [App08e]. I change Fenestra to use an NSMatrix in Section 22.3, *Using NSMatrix to Organize Buttons*, on page 289.

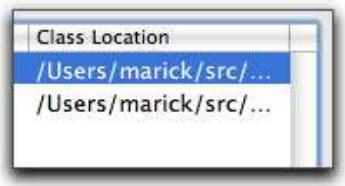
The Source column should contain names of Ruby files to load. We want to be able to change a cell value in three ways:

- Double-click it to open the usual file chooser panel.
- Drag a file from, say, the Finder, and drop it on the cell.
- Type in it, just like any other cell. To make that pleasant, though, we'll need to add a cell formatter. That's the topic of the next chapter.

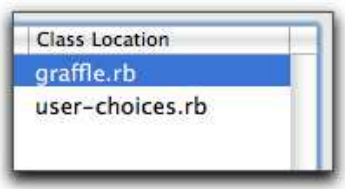
Chapter 18

A Formatter with Two Wrinkles

The Source column in the preference panel will contain full pathnames of files. That means long strings will need to fit into a small space. If we do nothing special, the Source column will end up looking like this:



That's not so inspiring. It would be better if the cell showed only the basename of the file, like this:

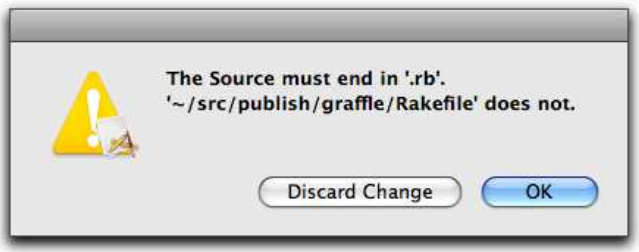


But if you edit the contents, you'd want to see the whole pathname:

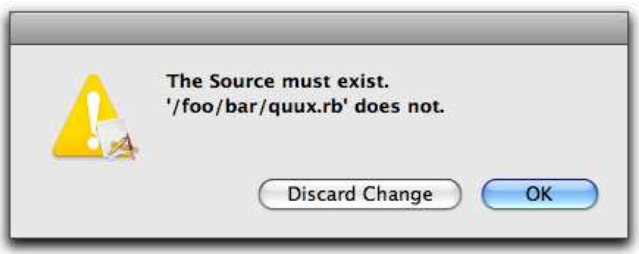
Otherwise, how would you correct a misspelled folder name?

Notice that the display abbreviates /Users/marick to ~. That seems convenient. I can think of two other conveniences to add:

- The cell should reject a pathname that doesn't end in .rb:



- To further guide against typos, Fenestra should check that the file actually exists. If not:



Making these changes gives me an excuse to explain how to call Objective-C methods that demand by-reference arguments.[1] I'll use tests in my explanation, so along the way I'll also explain another testing trick.

18.1 The Formatter Code

The formatter code, shown in Figure 18.1, on the following page, is similar to the BooleanCellFormatter from Section 13.3, *Formatters*, on page 172. At ❶ and ❷, it has the stringForObjectValue and getObjectValue_forString_
errorDescription methods you've already seen. Because we want the display during editing to look different from the normal display, we also define editingStringForObjectValue (at ❸).

1. You learned how to handle the *receipt* of by-reference arguments (using ObjcPtr) in Section 13.3, *Pointers to Pointers*, on page 172.

Download fenestra/table-formatter/app/prefs-window/formatters.rb

```
class BasenameFormatter < OSX::NSFormatter
  include OSX
  ib_outlet :cell

  def awakeFromNib; @cell.formatter = self;  end

❶ def stringForObjectValue(o)
    return '' if o.nil?
    File.basename(o)
  end

  # TODO: this is bad design - too many places in the code know a
  # little bit about the Source. That knowledge should be encapsulated,
  # and puts somewhere other than right next to the GUI.

❷ def getObjectValue_forString_errorDescription(objptr, s, errdesc)
    unless s.to_ruby =~ /\.rb$/
      return error("The Source must end in '.rb'. \n'#{s}' does not.",
                   errdesc)
    end

    unless file_exists?(s)
      return error("The Source must exist.\n'#{s}' does not.", errdesc)
    end

    objptr.assign(s)
    return true
  end

❸ def editingStringForObjectValue(o)
    return '' if o.nil?
❹   o.to_ns.stringByAbbreviatingWithTildeInPath
  end

❺ testable

  def file_exists?(name)
❻   name = name.stringByExpandingTildeInPath
❼   NSFileManager.defaultManager.fileExistsAtPath(name)
  end

  def error(msg, errdesc)
    errdesc.assign(msg) if errdesc
    false
  end
end
```

fenestra/table-formatter/app/prefs-window/formatters.rb

Figure 18.1: Formatting a pathname

I used one of Cocoa's built-in NSString methods to abbreviate (❹) and expand (❻) tildes in strings that represent paths. NSString has a huge number of methods; if there's no Ruby String method to do what you want, try looking at NSString's documentation.

NSFileManager (❼) is another handy class. If File or Dir doesn't have the method you want, look there.[2]

Line ❺ looks peculiar: there's a testable where you'd expect private, protected, or public. I'll explain that in Section 18.3, *Breaking Encapsulation in Tests*, on page 240.

18.2 Calling Methods That Take Reference Arguments

Two of the methods in BasenameFormatter can be tested using code that teaches you nothing new. For example:

Download fenestra/table-formatter/test/prefs-window/formatter-tests.rb

```
should "convert whole pathname to basename" do
  assert { @formatter.stringForObjectValue("/path/to/file.rb") ==
           "file.rb" }
end
```

getObjectValue_forString_errorDescription is tougher, though. Here is its Objective-C declaration:

```
- (BOOL) getObjectValue:(id *)anObject
         forString:(NSString *)string
         errorDescription:(NSString **)error
```

It has two by-reference arguments (pointers to pointers): anObject and error. In most cases, by-reference arguments are a way to kludge multiple return values into languages that don't allow them. Ruby *does* allow them, so RubyCocoa arranges for us to get all three results from getObjectValue_forString_errorDescription as return values. You can see that at ❶ in the following code. (For the moment, ignore the way the method is called.)

2. You're justified in suspecting me of either hypocrisy or carelessness here. In Section 6.4, *Using Nibs to Avoid Dependencies*, on page 90, I claimed an aversion to hard-coded class names in code and used nib loading to avoid them. I went to some trouble in Chapter 16, *Selections and Editing*, on page 202, to isolate PreferencesController from NSUserDefaultsController so that the former could be tested without worrying about the actual preferences file on disk. Yet file_exists? has a hard-coded use of NSFileManager, a use that could easily lead to fragile tests that break mysteriously when a file that's supposed to be on disk goes missing. In my defense, a better design would require more words of explanation and would show you nothing new.

Download fenestra/table-formatter/test/prefs-window/formatter-tests.rb

```
should "reject a file that doesn't exist" do
  name = "/does/not/exist.rb"
❶ accepted, obj, error_message =
      @formatter.objc_send('getObjectValue:forString', name,
                           'errorDescription')
  deny { accepted }
  assert { obj.nil? }
❷ assert { error_message.to_ruby  =~ /'#{name}'/ }
❸ assert { error_message.to_ruby  =~ /must exist/ }

end
```

The other two marked lines are interesting for two reasons:

- The return value they're working with is an NSString, but I converted it to a Ruby string before doing the regular expression comparison. If I hadn't, I would have gotten this warning:

  ```
  'NSString#=~' doesn't work correctly. Because it returns byte ↩
  indexes. Please use 'String#=~' instead.
  ```

  It also works to reverse the order and put the regexp on the left side of the **=~**.

- One of the banes of testing is the way tiny changes in output strings can break tests. A test whose failure tells you only that you put a period at the end of an output line—something you already knew—is likely to be more trouble than its worth over the long run. For that reason, I'll often use regular expressions to pick out only the bits of the output that are really important. I've used two separate ones here because I don't want even the order of the two snippets to matter.

Creating By-Reference Arguments

When I first thought of how I would call getObjectValue_forString_errorDescription, I imagined it would look like this:

```
@formatter.objc_send(:getObjectValue, ????,
                     :forString, name,
                     :errorDescription, ????)
```

What do you pass in for the by-reference arguments? My first thought was "Why, ObjcPtrs, of course," but that's incorrect. Instead, you simply *leave out* the by-reference arguments.

Leaving out the final one is easy:

```
@formatter.objc_send(:getObjectValue, ????,
                     :forString, name,
❶                    :errorDescription)
```

The earlier mystery argument is trickier. Let's start by writing the keywords surrounding it as strings rather than symbols:

```
@formatter.objc_send("getObjectValue", ????,
                     "forString", name,
                     :errorDescription)
```

Symbols, strings—they're all the same as far as objc_send is concerned. Now let's drop the mystery argument:

```
@formatter.objc_send("getObjectValue",
                     "forString", name,
                     :errorDescription)
```

But this is ambiguous. Is *"forString"* a keyword preceded by an omitted argument? Or is it, itself, the argument to *"getObjectValue"*? There's no way to tell for sure.

To disambiguate, the two keywords are concatenated, with a colon to mark the end of one and the beginning of the other:

```
@formatter.objc_send("getObjectValue:forString", name,
                     :errorDescription)
```

Nil Values

What if the user of getObjectValue_forString_errorDescription doesn't want an error description? In that case, passing a nil instructs the method not to create one. That looks like this:

Download fenestra/table-formatter/test/prefs-window/formatter-tests.rb

```
should "not return an error message unless one is asked for" do
  name = "invalid"
  accepted, obj = @formatter.objc_send('getObjectValue:forString', name,
                                       'errorDescription', nil)
  deny { accepted }
  assert { obj.nil? }
end
```

Notice that now there are only two return values.

More Details

The tests in rubycocoa-oddities/test/call-by-reference-tests.rb show more details about calling methods with by-reference arguments, including a remaining ambiguity that might trip you up.

18.3 Breaking Encapsulation in Tests

The BasenameFormatter has this utility method:

Download fenestra/table-formatter/app/prefs-window/formatters.rb

```
def file_exists?(name)
  name = name.stringByExpandingTildeInPath
  NSFileManager.defaultManager.fileExistsAtPath(name)
end
```

That simple little method provokes a tug-of-war between two traditions:

- The *object-oriented design* tradition says that file_exists? is not part of BasenameFormatter's public contract, so it should be made protected or private.
- The *testing* tradition says that it should be tested and that, most often, it's best if the tests call it directly. But tests can't do that if it's protected or private.

How do we reconcile the contradiction? The easiest thing to do is not to bother and just choose one of these solutions:

- Encapsulate file_exists?, and test it indirectly by trying to provoke getObjectValue_forString_errorDescription into calling file_exists? the way the test wants.
- Throw off the chains of encapsulation, make file_exists? public, and test it directly.

I confess that I'm fairly casual about encapsulation, so I often choose the second solution. But there are other choices:

- Make a subclass of BasenameFormatter that exposes private methods like file_exists?. Use that subclass in tests, but don't make it available to production code.
- Obey this common rule of thumb: *if it's hard to test, there's something wrong with it.*[3] So, you say you can't test validation code because it's buried inside BasenameFormatter? Extract a new class that's all about validation, make the validation methods public, test them, and have BasenameFormatter use an instance of the new class.

  I bet that's the best solution in this case. I'm already skeptical about a UI-centric class that has "business logic" smeared

3. I think I first heard that from Carl Erickson of Atomic Object.

throughout it. Tests—as they often do—could give me the push to make a better design, one with loosely coupled classes that each do one thing and do it well.

- Use a programming language with the usual public, protected, and private access types—but also a testable access type.

  After all, tests have a special relationship to the class's code, just like subclasses do. So, it makes sense for them to have a special kind of access, just like subclasses have.

  Of course, there is no such language.

Although the two-class solution is probably the best, I'm going to resist virtue in this case. Instead—despite the niggling problem that there's no such thing—I'm going to use testable access:

Download fenestra/table-formatter/app/prefs-window/formatters.rb

```
class BasenameFormatter < OSX::NSFormatter
  # ...

  testable

  def file_exists?(name)
    name = name.stringByExpandingTildeInPath
    NSFileManager.defaultManager.fileExistsAtPath(name)
  end
  # ...
end
```

In languages like Java, the tokens public, protected, and private are special keywords. In Ruby, they're just class methods, so it's easy to add a new one. testable gives the same access as protected but signals a special intent for the methods following it.

file_exists? is still behind an encapsulation boundary, but there's a mechanism for tests to subvert that boundary. Such a test is shown here:

Download fenestra/table-formatter/test/prefs-window/formatter-tests.rb

```
should "expand tildes when checking file existence" do
❶   @formatter.extend(Fenestrable)
❷   assert { @formatter.fenestra.file_exists?("~".to_ns) }
end
```

❶ A test that needs to can poke a hole through the encapsulation boundary by extending the object-under-test. That hole is an attribute named fenestra.

❷ One can then call encapsulated-but-testable methods through the hole.[4]

I'm explaining this testability support mainly so that you aren't puzzled when you come across it in my tests and code, but I also have a quiet hope that you'll love the idea so much that you'll use it in all your Ruby code. If so, you can find the source in app/util/testutil.rb in this and any following version of Fenestra.[5]

18.4 What Now?

Typing long pathnames into a text field is not the way to spend a lazy Sunday afternoon. OS X gives you better alternatives. In the next two chapters, we'll make Fenestra support them.

4. Actually, you can call any method. It wouldn't be hard to allow only methods marked testable to be called, but I haven't bothered.

5. I consider it a virtue of my solution that you need to add a testable declaration to the class under test—it constantly nags at you to reconsider a likely sloppy factoring of your classes. If you don't need or want such nagging, there are alternatives that don't change the class under test. The simplest is to call private methods with send. http://jasonrudolph.com/blog/2007/11/02/evan-phoenix-on-testing-private-methods-in-ruby/ and http://blog.jayfields.com/2007/11/ruby-testing-private-methods.html show different solutions. (Thanks to Jakub Suder for telling me of these links.)

Chapter 19

Picking Files with Open Panels

In this chapter, we'll change Fenestra so that double-clicking a Source cell opens Cocoa's standard Open panel. The chosen file will be put into user preferences.

19.1 NSOpenPanel

NSOpenPanel is the class for panels that let you choose one or more files. Conveniently enough, we can use irb to learn about it.

You create an NSOpenPanel with the openPanel method:

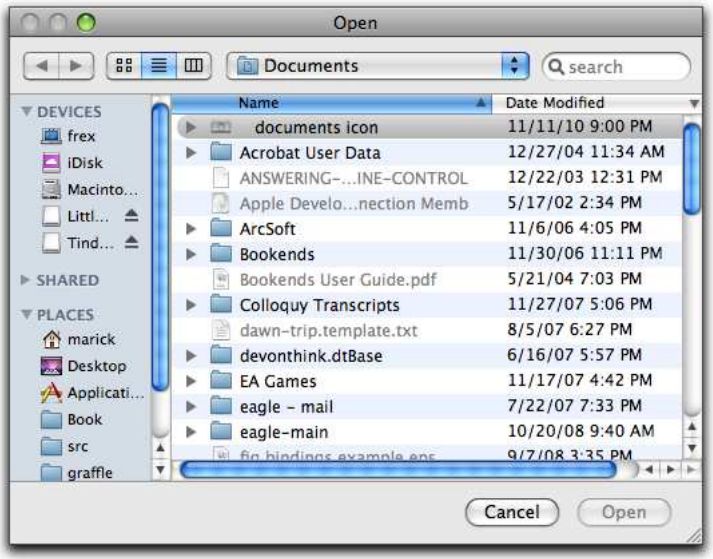
```
irb(main):001:0> require 'osx/cocoa'
=> true
irb(main):002:0> include OSX
=> Object
irb(main):003:0> panel = NSOpenPanel.openPanel
=> #<OSX::NSOpenPanel:0x296030 class='NSOpenPanel' id=0x1b3d1f0>
```

Although its name might suggest it, the openPanel method doesn't show the panel on the screen. To do that, you use a different method:

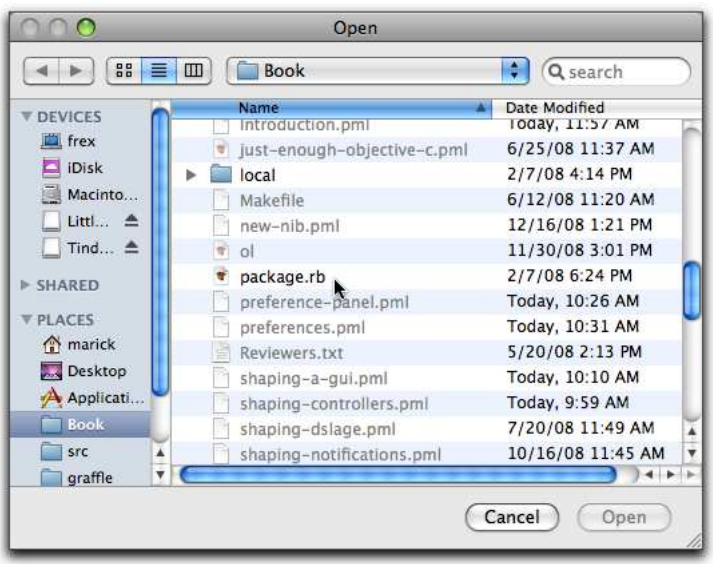
```
irb(main):004:0> panel.runModalForTypes(["rb"])
```

That method doesn't immediately return.

Instead, it pops up a panel much like this one:



For lack of a better choice, the Open panel shows the Documents folder. The folders show normally, but the actual files are grayed out. That's because we've restricted the selectable files to those ending in .rb. There happen to be no Ruby files in this folder, but I can find some in this book's root folder:



If I choose one and click the Open button, runModalForTypes will return this:

```
irb(main):005:0> panel.runModalForTypes(["rb"])
=> 1
```

That return value names the button that was clicked:

```
irb(main):006:0> NSOKButton
=> 1
```

Open panels can, by default, select multiple files. You can retrieve which ones were selected with the filenames method:

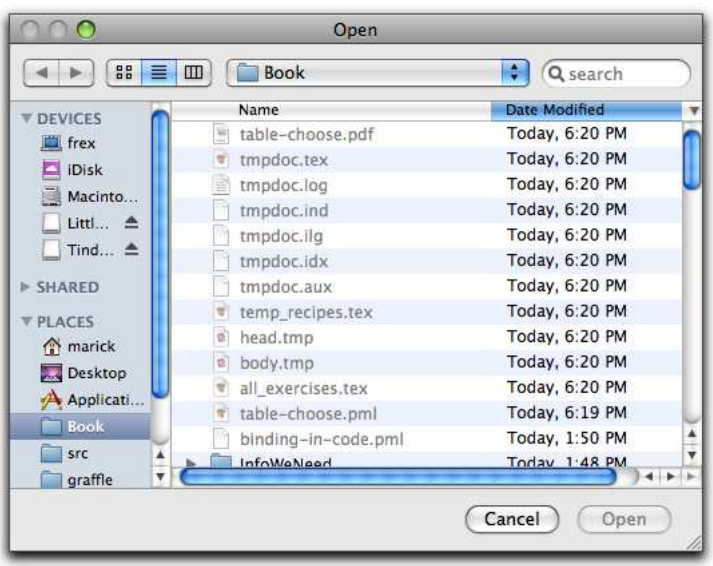
```
irb(main):007:0> panel.filenames
=> #<NSCFArray [#<NSPathStore2 "/Users/marick/.../Book/package.rb">]>
```

An NSPathStore2 looks like an exotic class, but it's just a subclass of NSString.

Open panels store their most recent folder in the app's preferences. You can see that in action by exiting irb, restarting it, creating a new panel, and running it. Like this:

```
irb(main):008:0> exit
$ irb
irb(main):001:0> panel = NSOpenPanel.openPanel
=> #<OSX::NSOpenPanel:0x296030 class='NSOpenPanel' id=0x1b3d2c0>
irb(main):002:0> panel.runModalForTypes(["rb"])
```

Once again, I'm in the Book folder:



A Note on Preferences

In this version of Fenestra, I changed the default value of a TranslatorPreference's source property from nil to "". I didn't have any problems with a legacy preferences file, but it might be prudent to remove ~/Library/Preferences/com.apple.rubycocoa/FenestraApp.plist.

Try This Yourself

Here's a list of changes you can make by finding and setting appropriate NSOpenPanel properties:

- Change the title of the panel to something other than Open.
- Change the Open button to say something else, like Choose.
- Can the panel's user choose more than one file? If so, restrict her to one file.

When looking for setters, note that NSOpenPanel is a subclass of NSSavePanel. Methods that you might expect to be described in NSOpenPanel's API documentation are actually described in the superclass's documentation.

19.2 A Design for Using NSOpenPanel in Fenestra

In the rest of this chapter, you'll change Fenestra so that a double-click in a Source cell will pop up an NSOpenPanel.

It will let the user choose a single Ruby file. When the user confirms the choice (rather than canceling), the new source will be stored in user preferences.

Begin with the version of Fenestra in fenestra/table-chooser-start. It has the tests but not the code to pass them. You can find code that passes them in fenestra/table-chooser-end.

When I originally worked on this chapter's Fenestra, I started by doing all the work in PreferencesController. Getting the manipulation of the panel correct in the same code that used its results started to feel con-

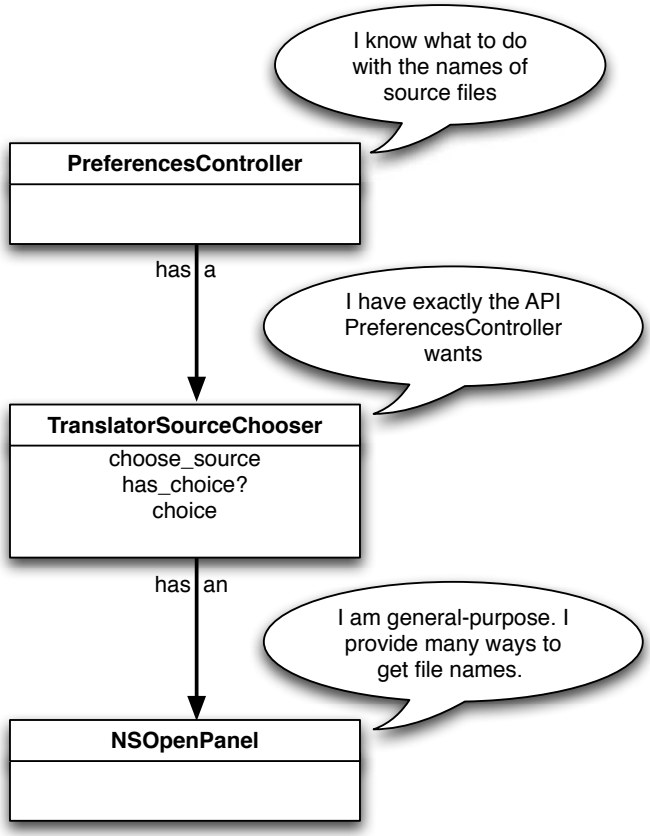


Figure 19.1: Adapting the general-purpose class

fusing and ugly, so I switched to what Abelson and Sussman[1] call "programming by wishful thinking." I coded PreferencesController against a file-choosing interface perfect for my needs, looked into the Apple API documentation to see whether that perfect interface already existed, found it didn't, sighed, and wrote an adapter class that implemented my perfect interface in terms of what I had available (NSOpenDialog). The result looks like Figure 19.1.

As I continued, though, the tests began to get harder to write. After my usual period of bullheadedly not listening to what they were telling me,

1. In their masterful *Structure and Interpretation of Computer Programs* [AS96].

I finally put the work on hold and took a long shower.[2] While scrubbing, I realized that the Open panel was another window on the screen, and therefore it should be backed by some controller object. That object should be more independent than the TranslatorSourceChooser in Figure 19.1, on the preceding page.

Making a controller for the NSOpenPanel made me think of the way the controllers in the main window communicate via notifications rather than by an asymmetrical call/return pattern. I decided to do the same in the preference panel. That idea is shown in Figure 19.2, on the next page. Is it the *right* idea? We'll see. . .

19.3 Try This Yourself: PreferencesController Tests

The PreferencesController will handle double-clicks in the table.

I see four situations to worry about: initializing, handling double-clicks outside a source cell, sending a notification after a double-click inside a source cell, and receiving a notification that a new source has been chosen. I'll start this section by giving you both tests and the code that passes them, but I'll soon let you write the code.

Initialization

Double-clicks use the same target-action idiom as single clicks, so we need to initialize the NSTableView to send its double-clicks to the controller. Here are the two tests that describe that:

Download fenestra/table-chooser-start/test/prefs-window/prefs-change-source-tests.rb

```
context 'initialization' do
  should "make the preferences controller the target of the table" do
    assert { @table.target == @sut }
  end

  should "make :doubleClick the action taken on doubleclick" do
    assert { @table.doubleAction == 'doubleClick:' }
  end
end
```

2. The brilliant Guy Steele once said that he gets all his good ideas in the shower. Me too. But he must take a *lot* more showers than I do.

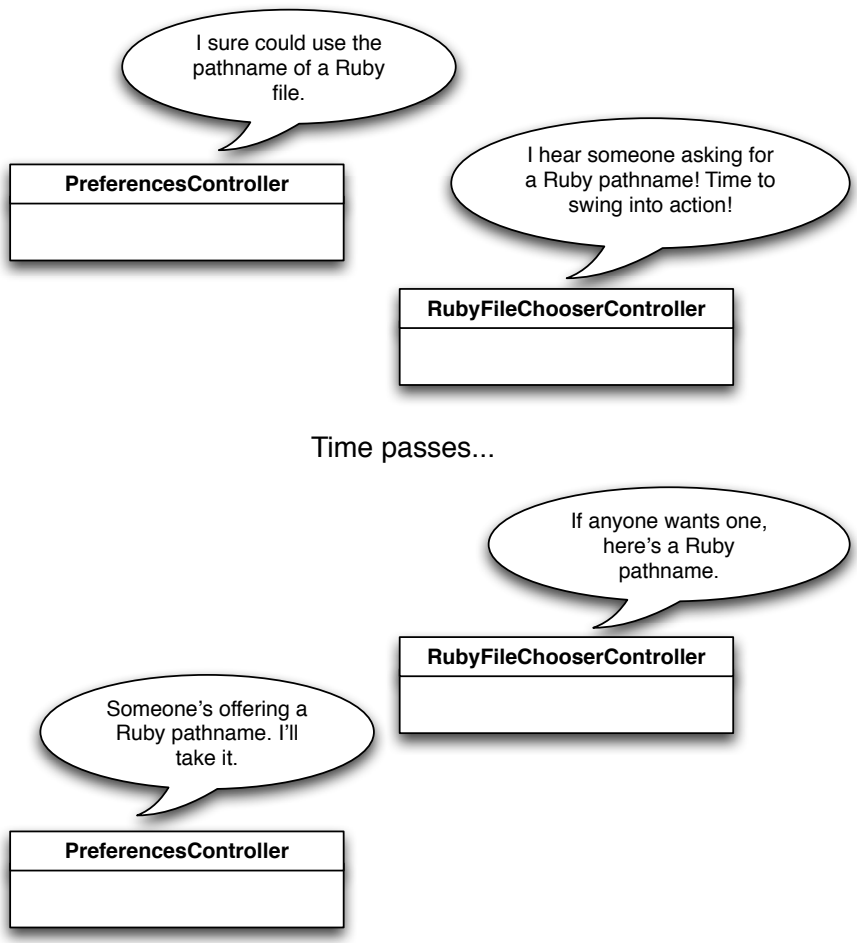


Figure 19.2: Peers instead of caller and called

The work is done in awakeFromNib:

Download fenestra/table-chooser-start/app/prefs-window/PreferencesController.rb

```
def awakeFromNib
  # ...
  @table.target = self
  @table.doubleAction = "doubleClick:"
end
```

Notice that the doubleAction is a string naming a Ruby method in the Objective-C style. When the doubleAction method is called, it will be given the @table as its argument.

Double-Clicks Outside a Source Cell

Most people, I think, like to implement the "happy path" first. Their first tests check that the app does the right thing in simple cases. After that, they build up to more complex cases, including error cases. Ever the contrarian, I'll often do the error cases first. Like a lot of people, I can't stop myself from thinking of error handling as less important. It's code that prevents value from being lost (by preventing something bad from happening), and I'd rather write code that produces value. So when I add error handling to a "finished" feature, I'm eager to get it over with and move on. That makes me do a worse job of imagining error cases.[3]

I want to prevent the code from making two types of errors:

- The user double-clicks a Display Name cell and—oops!—the file chooser panel pops up. Instead, the non-Source cells should keep their normal behavior: a double-click should start ordinary text-field editing.[4]
- The user clicks somewhere in the table header, below the final row, or somewhere else that's not in any cell and—oops!—the file chooser panel pops up.

3. The same "just let me get this over with" psychology applies to adding tests after the code is done, which is one of the reasons writing the tests first works so much better.

4. The user can still edit the Source text field in-place by first clicking to select its line and then clicking again on the Source text to begin editing. That—two clicks far enough apart that they're not interpreted as a double-click—can also be used for the non-Source text fields.

Retaining Behavior for Other Columns

Here's a test that checks whether my first fear comes true:

Download fenestra/table-chooser-start/test/prefs-window/prefs-change-source-tests.rb

```
context "clicking on non-source column" do
  should 'still perform normal editing' do
❶   during_doubleclick_on(@display_name_column_index, 1).behold! {
❷     @table.should_receive(:editColumn_row_withEvent_select, 4).once.
            with(@display_name_column_index, 1, any, 1)
❸     watchers_are_notified.never
    }
❹   assert { @original_sources == self.current_sources }
  end
end
```

❶ during_doubleclick_on simulates a double-click in the first row's Display Name cell. It does that by calling the PreferencesController's doubleClick method after arranging for NSTableView's clickedColumn and clickedRow methods to return the given values.

❷ Because the cell "clicked" is in the Display Name column, the test expects the table to receive the edit message you saw back in Section 16.2, *Working with an Uncooperative Control*, on page 212.

❸ It's not enough for the code to do the right thing, it must also *refrain* from doing the *wrong* thing. In this case, the "wrong thing" would be to send out an "I sure could use the pathname of a Ruby file" notification. So, I explicitly check that no notification is sent. (See the sidebar on the next page.)

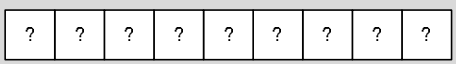
❹ For good measure, the test also checks that both the arrangedObjects and NSUserDefaultsController contents haven't changed.

Among people who are used to writing tests before code, it's common to pass the first test with code that'll obviously fail on the very next one.

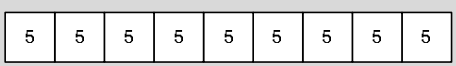
Initialize and Check the Background

A common programmer mistake is to do more than is right. For example, back in my days as a C programmer, I used to look for the following kind of bug:

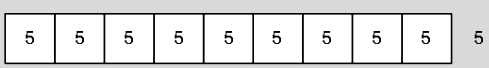
1. We began with an array of, say, nine elements. Their contents don't matter:



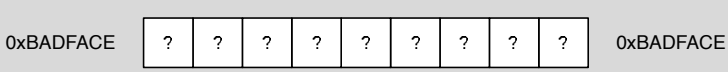
2. The code was supposed to fill the array with the number 5:



3. It did that. . . but it also scribbled past the end of the array:



4. Especially in C, these can be horrible bugs to find. So, I started thinking of the data that was supposed to change as the "foreground" and the rest as "background." (Think of cheaper cartoons where the characters move but the background stays unchanged.) In the arrange part of my tests, I would explicitly put known elements in the background:



   Then the assert part would check that the background hadn't been overwritten.

This particular bug is less common in this day of iterators, but the analogy still holds. For this chapter, the background is the quiet computational void that should not be disturbed by notifications or stray message sends.

PreferencesController already has a shorthand method for editing cells, edit_cell_at, so we can just call it:[5]

Download fenestra/table-chooser-start/app/prefs-window/PreferencesController.rb

```
def doubleClick(sender)
  edit_cell_at(@table.clickedColumn, @table.clickedRow)
end

protected

def edit_cell_at(col, row)
  select_row_with_index(row)

  @table.objc_send(:editColumn, col,
                   :row, row,
                   :withEvent, nil,
                   :select, true)
end
```

Someone who's truly hard-core might even replace the calls to clickedColumn and clickedRow with 0 (the index of the Display Name column) and 1. After all, there's no evidence—in the form of a test—that the code will ever have to handle any other values. I won't go to that extreme, though it can be an interesting one to practice on.

Clicks in the Wrong Places

Now I'd like you to take over. Change should_eventually to should in the following tests, and make them pass:

Download fenestra/table-chooser-start/test/prefs-window/prefs-change-source-tests.rb

```
  context "clicking outside normal bounds" do
    should_eventually 'do nothing (if out of row bounds)' do
❶     during_doubleclick_on(@source_column_index, -1).behold! {
❷       nothing_happens
      }
      assert { @original_sources == self.current_sources }
    end

    should_eventually 'do nothing (if out of column bounds)' do
❸     during_doubleclick_on(-1, 1).behold! {
❹       nothing_happens
      }
      assert { @original_sources == self.current_sources }
    end
  end
end
```

5. Notice that using the shorthand method reveals a bug in the test. The test failed to specify that the row has to be selected. I recommend you fix the test.

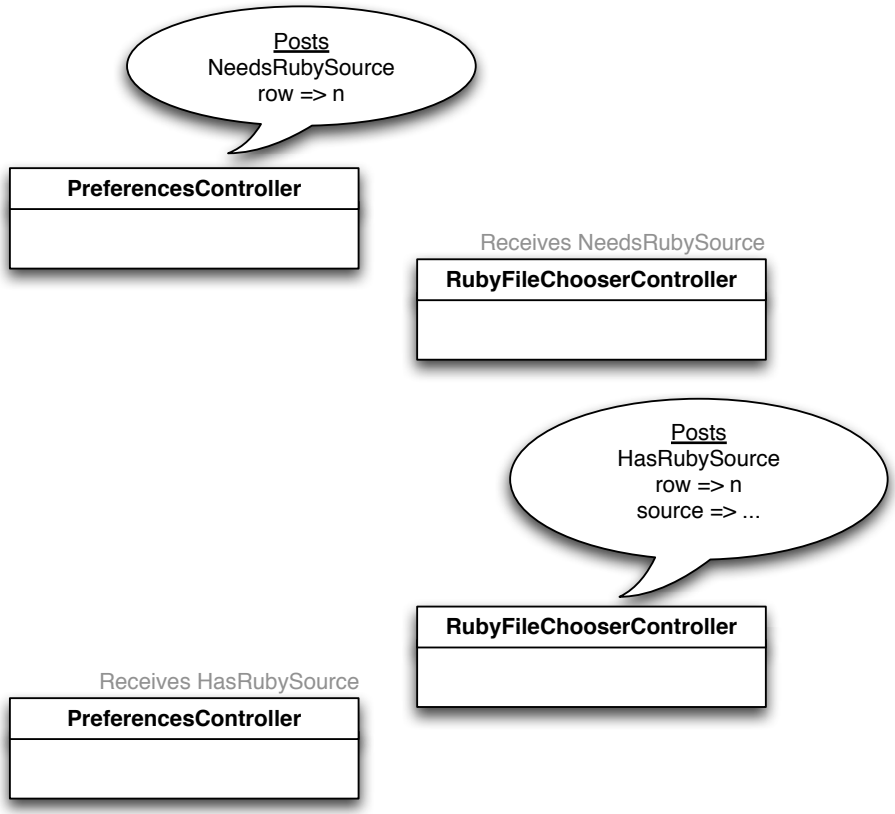


Figure 19.3: Controller notifications

The -1 values at ❶ and ❸ are key to the tests. -1 is the value NSTableView's clickedColumn, and clickedRow methods return if the click was outside column or row bounds. nothing_happens (at ❷ and ❹) asserts that no notifications are posted and the table cell is not edited.[6]

You can find my solution for both these tests and the following PreferencesController tests in Figure 19.5, on page 259.

6. nothing_happens will not notice if your code sets the table's intercellSpacing to some huge value, just as the test in the sidebar wouldn't notice if code scribbles somewhere other than at the boundaries of the array. In error prevention and detection, you have to play the odds.

Deciding on Vocabulary

Now we get to the point of it all: communication between this PreferencesController and the RubyFileChooserController (that we haven't started yet).

Figure 19.3, on the previous page, is a variant of Figure 19.2, on page 249. The new version replaces English text with descriptions of NSNotifications. The first speech bubble says that the PreferencesController will send a notification named NeedsRubySource whose userInfo will contain a row number. The RubyFileChooserController will pass that row number back, along with the pathname of a Ruby file.[7]

Double-Clicks Within a Source Cell

This test implements the first speech bubble we see in Figure 19.3, on the previous page:

Download fenestra/table-chooser-start/test/prefs-window/prefs-change-source-tests.rb

```
context "clicking in Source cell of a row" do
  should_eventually "post a notification asking for a Ruby file" do
    during_doubleclick_on(@source_column_index, 0).behold! {
      watchers_are_notified.once.
                            with(on { | notification |
                                   row = notification.userInfo[:row]
                                   name = notification.name
                                   0 == row && name == NeedsRubySource
                                 })
      no_editing_starts
    }
  end
end
```

Make it fail (by changing should_eventually), and then make it pass. You can use the post method described in Section 8.3, *Shorthand for Posting Notifications*, on page 103.

Receiving a Notification

Occasionally, the PreferencesController will receive a notification of a new source.

7. It's good to worry about a possible bug here: what if the rows are rearranged between the time NeedsRubySource is sent and HasRubySource is received? I believe that's impossible.

This test describes how it should be handled:

Download fenestra/table-chooser-start/test/prefs-window/prefs-change-source-tests.rb

```
context "receiving a source file notification" do
  should_eventually "store the new source in preferences" do
    new_file = "/tmp/mumble.rb"

    some_object_announces(HasRubySource,
                          { :row => 0,
                            :source => new_file })

    expected = [new_file] + @original_sources[1..-1]
    assert { self.current_sources == expected }
  end
end
```

Make it pass by filling in this skeleton:

Download fenestra/table-chooser-start/app/prefs-window/PreferencesController.rb

```
on_local_notification HasRubySource do | notification |
end
```

on_local_notification was described in Section 8.1, *A DSL for Notifications*, on page 101.

19.4 Try This Yourself: The NSOpenPanel Controller

Figure 19.4, on the following page, shows the tests that need to pass to create a working RubyFileChooserController. Make them pass. The code to start with is in app/prefs-window/RubyFileChooserController.rb. My solution is in Figure 19.6, on page 260.

Again, What Are We Testing?

It's important to realize that these tests document my *assumptions* about how NSOpenPanel behaves: that it never returns anything except either NSOKButton or NSCancelButton; that the way to restrict it to only Ruby files is with the type *"rb"*, not *".rb"*; and so on.

How did I come to make those particular assumptions? By doing pretty much what you did at the beginning of this chapter: I skimmed over the API documentation, found likely methods, and tried them out in irb. I prefer that to just relying on the documentation.

Download fenestra/table-chooser-start/test/prefs-window/file-chooser-tests.rb

```
should_eventually "respond to NeedsRubySource by starting chooser" do
  during {
    some_object_announces(NeedsRubySource)
  }.behold! {
    @chooser_panel.should_receive(:runModalForTypes, 1).once
  }
end

should_eventually "restrict chooseable files to ruby files" do
  during {
    some_object_announces(NeedsRubySource)
  }.behold! {
    @chooser_panel.should_receive(:runModalForTypes, 1).once.
                   with(['rb'])
  }
end

should_eventually "do nothing if panel cancels" do
  during {
    some_object_announces(NeedsRubySource)
  }.behold! {
    @chooser_panel.should_receive(:runModalForTypes, 1).once.
                   and_return(NSCancelButton)
    watchers_are_notified.never
  }
end

context "when file is chosen," do
  should_eventually "announce HasRubySource with source and row" do
    during {
      some_object_announces(NeedsRubySource, { :row => 1000 })
    }.behold! {
      chosen_source = 'path/to/ruby/file.rb'

      @chooser_panel.should_receive(:runModalForTypes, 1).once.
                     and_return(NSOKButton)
      @chooser_panel.should_receive(:filename).once.
                     and_return(chosen_source)

      expected = this_notification(HasRubySource,
                                   @sut,
                                   { :source => chosen_source,
                                     :row => 1000 })
      watchers_are_notified.once.with(expected)
    }
  end
end
```

fenestra/table-chooser-start/test/prefs-window/file-chooser-tests.rb

Figure 19.4: A description of RubyFileChoserController

There's many a slip 'twixt the cup and the lip, though, and also between the mind and the test case. So, it doesn't hurt to check the assumptions again by using the should statements as a checklist to follow when manually trying the new feature that crucial first time.

But take care not to *restrict* yourself to the checklist. Explore. For example, I discovered by exploring that double-clicking a Favorite cell gives me a text edit field for a radio button's label (which we made empty on page 232). It looks as if double-clicking should be handled specially for the Favorite column as well.

19.5 What Now?

It's time to finish up the preference panel by allowing drag and drop.

Download fenestra/table-chooser-end/app/prefs-window/PreferencesController.rb

```
class PreferencesController < OSX::NSArrayController

  def doubleClick(sender)
    return if @table.clickedRow == -1
    return if @table.clickedColumn == -1
    unless COLUMN_IDS[@table.clickedColumn] == :source
      edit_cell_at(@table.clickedColumn, @table.clickedRow)
      return
    end
    post(NeedsRubySource, :row => @table.clickedRow)
  end

  on_local_notification HasRubySource do | notification |
    row = notification[:row]
    update_source_at_clicked_index(row, notification['source'])
  end

  protected

  def update_source_at_clicked_index(index, new_source)
    update_changed_object_at_index(index) do | pref |
      pref.source = new_source
    end
  end

  def update_changed_object_at_index(index)
    obj = arrangedObjects[index]
    removeObjectAtArrangedObjectIndex(index)
    yield(obj)
    insertObject_atArrangedObjectIndex(obj, index)
  end
end
```

fenestra/table-chooser-end/app/prefs-window/PreferencesController.rb

Figure 19.5: PreferencesController code for file choosing

Download fenestra/table-chooser-end/app/prefs-window/RubyFileChooserController.rb

```
class RubyFileChooserController < Controller

  def init
    initWithPanel(NSOpenPanel.openPanel)
  end

  def initWithPanel(panel)
    super_init
    panel.title = "Choose a Ruby File"
    panel.prompt = "Choose"
    panel.allowsMultipleSelection = false
    @panel = panel
    self
  end

  on_local_notification NeedsRubySource do | notification |
    return unless @panel.runModalForTypes(['rb']) == NSOKButton
    post(HasRubySource,
         :row => notification[:row], :source => @panel.filename)
  end

  testable

  attr_reader :panel
end
```

fenestra/table-chooser-end/app/prefs-window/RubyFileChooserController.rb

Figure 19.6: RubyFileChooserController

Chapter 20

Drag and Drop

Cocoa tables support two kinds of drag and drop. In one, people drag a whole row's worth of data onto the table and drop it. That's probably most common, but it's not the one we want. We want to drop pathnames into a table cell. I think that's fortunate, because that forces us to use a generic API that works for all views, rather than one that works only for NSTableView classes.

In this chapter, you'll begin with the source in fenestra/table-drag-start and convert it into code that passes its tests. You can see such code in fenestra/table-drag-end.

20.1 How Drag and Drop Works

As a dragged item enters, the view is sent a draggingEntered message. As long as the cursor stays inside it, the view is periodically sent a draggingUpdated method. Each of these methods returns a value telling the caller what will happen if a drop is attempted. For example, NSDragOperationNone means that the drop will fail.

When the user drops, the view is sent prepareForDragOperation. The view can signal that it refuses to cooperate by returning false. If it cooperates, though, it's then sent performDragOperation. If the method is successful, it should return true, whereupon the view will be sent concludeDragOperation.

If the user decides not to drop in this view and moves the cursor out of it, it will receive draggingEnded.[1]

1. You can learn more about this process in Apple's *Drag and Drop Programming Topics*

Pasteboards

Apps use *pasteboards* to stash data (or promises of data) or to pick up data that other apps have stashed. One pasteboard is used to store data copied with Command-C. A different one is used to store data that's being dragged.

The same "thing" can be put on the pasteboard in several different formats. For example, an application providing the name of a file can provide it as a string, as a single element in an array of strings, and as an NSURL object. The object removing the file from the pasteboard will use whichever format it prefers.

You can learn more about this process in Apple's *Pasteboard Programming Topics for Cocoa* [App08t].

NSDraggingInfo

The different drag methods—draggingUpdated and friends—are passed an NSDraggingInfo object that the view can ask questions of.[2] The most important bit of information is the pasteboard associated with the drag. (Although there's a pasteboard specifically reserved for dragging, there's no guarantee that the source app uses it.) It also includes a list of operations the source supports (copy, link, move, and so on) and the location of the cursor.

Coordinate Systems

Although the drag messages are received by a view, the NSDraggingInfo's location is in the coordinate system of the view's *window*. So if the view cares about the cursor's location within itself, it has to transform the value to its own coordinate system. The relationship between coordinate systems is shown in Figure 20.1, on the next page.

In the window's coordinate system, the origin is at the bottom left. In views, the coordinate system can be either at the bottom left or at the

for Cocoa [App08k].

2. Although the name looks like one, NSDraggingInfo isn't a class. Instead, it's the name of a list of methods, what Objective-C calls a *protocol*. A protocol is akin to an interface in Java, except that protocols can be "informal," meaning that the Objective-C compiler doesn't complain if you leave some of the methods unimplemented. For our purposes, a protocol is a name you can type into the search window in Xcode's documentation browser to find out about a group of related methods. About the only time you'll notice a protocol isn't a class is when you try to subclass it or alloc it in irb.

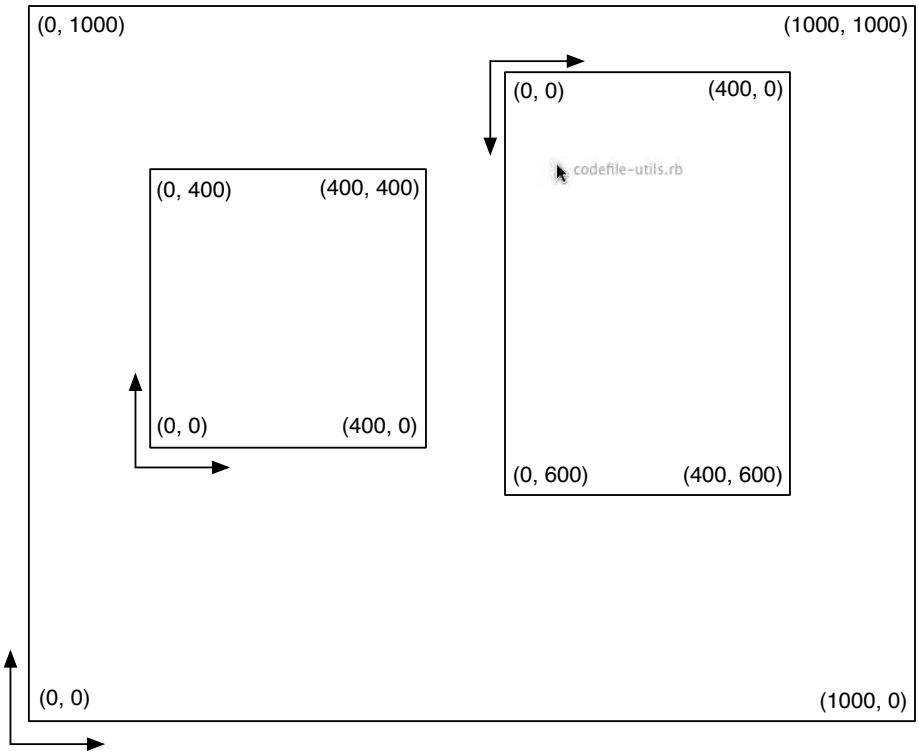


Figure 20.1: Coordinate systems

top left, depending on whether the view is more naturally thought of as growing up or down. Tables grow down.

In the figure, pretend that the rightmost view is a table view. The point of the cursor appears to be at about (729, 784) in the window's coordinate system. In the table's coordinate system, it's at (172, 229).

Translating among coordinate systems is an annoyance. Fortunately, you don't have to do the calculations yourself—the convertPoint_fromView and convertPoint_toView methods do that. (See Section 20.4, *NSTableView Patches*, on page 266, for an example.)

20.2 Designing the GUI

In Fenestra, the user should not be able to drop a filename anywhere other than a Source cell. The user interface should make that

restriction visually obvious. My first thought was to have the cursor change to be the "forbidden" cursor you see in some applications, like the Finder. Here's part of a Finder window that shows it:

As far as I can tell, though, that cursor design is not available in Cocoa. It's found only in the older framework named Carbon. So, after some trial and error, I decided to signal that a drop would work by highlighting the current row and changing the cursor to the "copy" cursor. That looks like this:

Since my UI design skills are laughable, not laudable, feel free to improve on this interface. The tests, however, assume it.

20.3 A Template for the Solution

Since dragging and dropping works by sending particular messages to a view, it makes sense to subclass NSTableView with our own class that implements them. See Figure 20.2, on the next page.

I didn't include skeletons for prepareForDragOperation and concludeDragOperation because there's nothing to be done in either case.

Download fenestra/table-drag-start/app/prefs-window/PreferencesTableView.rb

```
class PreferencesTableView < OSX::NSTableView

  def awakeFromNib
    # ...
  end

  def draggingEntered(info)
    # ...
  end

  def draggingUpdated(info)
    # ...
  end

  def performDragOperation(info)
    # ...
  end

  def draggingExited(info)
    # ...
  end

end
```

fenestra/table-drag-start/app/prefs-window/PreferencesTableView.rb

Figure 20.2: An NSTableView subclass

For this new class to be used, it needs to be added to the nib by changing the table's class from NSTableView to PreferencesTableView. I've already done that for you.

20.4 Utility Classes and Modules

When I originally implemented the UI design, I began by writing tests for the new PreferencesTableView class. As I did so, I discovered groups of methods that seemed to belong in their own class or module. By that, I mean that they seemed to share a common role, a role I could name with a noun phrase like "NSTableView patches" or "cell-oriented location information." In this section, I'll show them to you so that you can use them when you make tests pass.

Download fenestra/table-drag-start/app/util/NSTableViewPatches.rb

```
module NSTableViewPatches
  include OSX

❶   def cell_for_window_point(window_point)
    table_point = convertPoint_fromView(window_point, nil)
    column_index = columnAtPoint(table_point)
    row_index = rowAtPoint(table_point)
    return column_index, row_index
  end

  def row_for_window_point(window_point)
    ignored, row_index = cell_for_window_point(window_point)
    row_index
  end

  def select_row(index)
    row_set = NSIndexSet.indexSetWithIndex(index)
    selectRowIndexes_byExtendingSelection(row_set, false)
  end

  def identifier_for_column_index(column_index)
    tableColumns[column_index].identifier
  end
end
```

fenestra/table-drag-start/app/util/NSTableViewPatches.rb

Figure 20.3: NSTableView patches

NSTableView Patches

NSTableViewPatches is a module that adds some convenience methods to Cocoa's NSTableView. See Figure 20.3.

The only method with an interesting implementation is cell_for_window_point (at ❶). It shows how you convert from window coordinates to table indexes. convertPoint_fromView converts into self's coordinate system (in this case, into an NSTableView's). The second argument is often a containing view. In our case, we want it to be the containing window—that's what nil means.

Once we have a point in view coordinates, columnAtPoint and rowAtPoint give us indexes.

Download fenestra/table-drag-start/app/util/CellOrientedDraggingInfo.rb

```
class CellOrientedDraggingInfo < OSX::NSObject
  include OSX

❶   def column; colrow[0]; end
❷   def row; colrow[1]; end
❸   def column_id; table.identifier_for_column_index(column); end

❹   def initWithTable_rawInfo(table, raw_info)
      @table = table
      @raw_info = raw_info
      init
    end

    private
❺   attr_reader :table, :raw_info

    def colrow
❻     table.cell_for_window_point(raw_info.draggingLocation)
    end
end
```

fenestra/table-drag-start/app/util/CellOrientedDraggingInfo.rb

Figure 20.4: Cell-oriented location information

Cell-Oriented Drag Info

Because I hate and fear coordinate transformations, I'd like the drag-and-drop code to pretend they don't exist. That means that I'd like to hide the NSDraggingInfo object I get from Cocoa behind an object that identifies locations with row and column indexes. CellOrientedDraggingInfo, shown in Figure 20.4, is that object.

❶-❸ These three methods are the point of the class. They let dragging info be retrieved in the form of a row index, a column index, or a column identifier.

❹ Because the coordinates in an NSDraggingInfo have to be transformed in the context of a particular view (in this case, a PreferencesTableView), this object needs access to both the view and the NSDraggingInfo.

❺ These two accessors are encapsulated because they're the details this class was created to hide.

❻ This helper method seems ridiculously circular. Although this class exists to provide a detail-hiding interface to PreferencesTableView, it actually delegates the important work to... PreferencesTableView. Why?

Suppose cell_for_window_point were implemented in CellOrientedDraggingInfo. It would look something like this:

```
def cell_for_window_point(window_point)
  table_point = @table.convertPoint_fromView(window_point, nil)
  column_index = @table.columnAtPoint(table_point)
  row_index = @table.rowAtPoint(table_point)
  return column_index, row_index
end
```

That perfectly matches Kent Beck and Martin Fowler's definition of *feature envy*: "A method that seems more interested in a class other than the one it actually is in."[3] The solution to feature envy is to move the method where it wants to be. I've squared this circle—"Keep it out of the PreferencesTableView!" "No, put it in the PreferencesTableView!"—by putting it in NSTableViewPatches, which PreferencesTableView will include. Because of that, a reader of the PreferencesTableView source can remain blissfully ignorant of the coordinate-munging going on behind the scenes.

20.5 Try This Yourself: Lively Dragging Info

Let's exercise our wishful thinking muscles (explained on page 246). PreferencesTableView is going to receive a packet of information from Cocoa describing something a user wants to drop. That packet is a *dumb object* (more politely described as a *data object*). It doesn't *do* anything—it just sits there, waiting for us to poke around inside it. That's contrary to the whole point of object-oriented programming. Objects are supposed to be active: lively packets of data and methods that other objects can delegate work to.

I've hidden the NSDraggingInfo inside a CellOrientedDraggingInfo, but the latter is no smarter than the former. So, let's make a subclass with a tiny bit of smarts: the ability to decide by itself whether a drop is appropriate.

3. In Fowler's *Refactoring* [FBB+99], p. 80.

It would be used like this:

Download fenestra/table-drag-start/app/prefs-window/PreferencesTableView.rb

```
def evaluate_location(tailored_info)
  if tailored_info.drop_would_work?
    # ...
  else
    # ...
  end
end
```

Here's the skeleton of the new class:

Download fenestra/table-drag-start/app/prefs-window/PrefsTableDraggingInfo.rb

```
class PrefsTableDraggingInfo < CellOrientedDraggingInfo
  def drop_would_work?
    # ...
  end

  def pathname
    # ...
  end

  def pathnames
    # ...
  end
end
```

I've added pathname and pathnames methods because they'll be useful in the implementation of drop_would_work? and also inside PreferencesTableView.

Warm-up: Pathname Methods

I suggest you start implementing PrefsTableDraggingInfo with the pathname methods. That'll teach you about the structure of the pasteboard info that an NSDraggingInfo contains. Tests to drive your implementation are shown in Figure 20.5, on the next page.

❶ rubycocoa_flexmock is the method that creates the kind of object that accepts should_receive. With this single string argument (optional), rubycocoa_flexmock creates an NSObject that responds to only enough methods to make should_expect work. The string is used in error messages.

❷ This line shows that rubycocoa_flexmock can also be given a Class as an argument. (That argument could be followed by a name for error messages, but I didn't bother.) In this case, the resulting

Download fenestra/table-drag-start/test/prefs-window/prefs-table-dragging-info-tests.rb

```
def setup
❶  @raw_info = rubycocoa_flexmock("raw dragging info")
❷  @sut = rubycocoa_flexmock(PrefsTableDraggingInfo) do | klass |
     klass.alloc.objc_send(:initWithTable, :unused,
                           :rawInfo, @raw_info)
   end
end

context "pathnames method" do
  should_eventually "provide pathnames found in dragging info" do
    during {
❸     @sut.pathnames
    }.behold! {
      @raw_info.should_receive(:draggingPasteboard, 0).once.
                and_return(pasteboard_with_files(*some_paths))
    }
❹   assert { @result == some_paths }
  end
end

context "pathname method" do
  should_eventually "provide only pathname found in dragging info" do
    # text omitted from figure - almost the same as previous test
  end
end
```

fenestra/table-drag-start/test/prefs-window/prefs-table-dragging-info-tests.rb

Figure 20.5: Tests about extracting pathnames from a pasteboard

object will act as if it is a PrefsTableDraggingInfo, except that its methods can be overridden with should_expect.

By default, rubycocoa_flexmock creates the object with alloc and initializes it with init. A block argument, as shown following this line, allows custom creation or initialization.[4]

It's unusual to "flexmock" the very class you're testing. Why am I doing that? The class we're testing, PrefsTableDraggingInfo, depends on another class, CellOrientedDraggingInfo. It depends on it as a subclass, rather than by having a variable that points to an instance of it, but there's one principle that applies to both cases:

4. For all the details on rubycocoa_flexmock, see the tests in sandbox/test/mock-talk-tests.rb. rubycocoa_flexmock is built on top of the FlexMock gem, and you can find its documentation at http://flexmock.rubyforge.org.

don't go to extreme lengths just to be able to say that you're testing with a real object.

Consider that although these particular tests don't call the superclass's row method, later ones do. (You'll see them on page 274.) row needs both an NSDraggingInfo and a PreferencesTableView to work on. For the conversion from window coordinates to row number to work, that PreferencesTableView has to be contained within an NSWindow. . . wait. Exactly why am I creating all these objects? To use—and, incidentally, test—one tiny method. Although I didn't show them to you, it already has tests. So, the incidental testing is going to add little to my confidence that row works.

Because the cost/benefit ratio doesn't look so good to me, I'll fake row in the later tests. That lets me initialize the @sut with some dummy argument instead of a real PreferencesTableView. That dummy argument is the symbol :unused in the line after ❷.

❸ Given that the point of the test is to check the return value from pathnames, it's a bit peculiar that the return value appears to be just thrown away. I could have written the following:

```
during {
  assert { some_paths == @sut.pathnames }
}.behold {
  ...
}
```

But that looks awkward to me because it breaks the connection between the linear structure of the test code and the time-order of the running test: first you do something in the during block, during which something happens as described in the behold! block, but then you flip backward to see what happens after the method-under-test runs.

I'd rather have the assertion after the during-behold! construct, but there's no way of getting the value into the assertion that isn't at least somewhat awkward. The least ugly way I've thought of is to have during stash the return value of its block in @result and use that variable later, as shown at ❹. If you think that's an abominable choice, remember: hate the sin, not the sinner.

missing code?

You can drag many other things than pathnames: arbitrary strings, images, RTF files, colors, and so forth. Why is there no

method that checks whether the NSPasteboard really contains pathnames? That's because the sole client of this class arranges for it to be used only on NSDraggingInfo that comes from dragging files. (You'll see how it does that at line ❷ in Figure 20.8, on page 276.) We could have a lively debate about whether PrefsTableDraggingInfo should redundantly check what PreferencesTableView supposedly ensures. In the interest of brevity, I'm coming down on the "once and only once" side of that debate. Meanwhile, it rages on everywhere on the Internet.

Make the tests pass. If you haven't already been doing it, I recommend changing one should_eventually to should, watching the test fail, making it pass, and then moving to the next should_eventually.

Hints

The API documentation for NSDraggingInfo and NSPasteboard should give you the information you need.

Be sure to take a look at the test helper method pasteboard_with_files. It shows that the pasteboard is expected to contain data of NSFilenamesPboardType, so that's the type of data you'll need to check that it contains.

My solution is in the bottom part of Figure 20.6, on the following page.

Will a Drop Work?

You now have enough helper methods to make drop_would_work? a terse, to-the-point method. The tests that'll drive your solution are shown in Figure 20.7, on page 274.

❶ The setup block is an example of dragging info that Fenestra should accept: the cursor is inside one of the table rows and inside the Source column, plus the item being dragged is a single Ruby file. The tag by_default that ends each "sentence" means that the expectation is used in the absence of any nondefault one for the method named.

❷, ❸ These tests (and others like them) override exactly one of the defaults to produce a minimal case where drop_would_work? should be false. For example, ❷ overrides the default row value of 0 with -1, which is the value Cocoa uses for "not on a row."

In testing, picking minimal error or exception cases is important. If a test case contained two reasons the drop should be rejected, code could pass it by checking only one.

Download fenestra/table-drag-end/app/prefs-window/PrefsTableDraggingInfo.rb

```
class PrefsTableDraggingInfo < CellOrientedDraggingInfo

  def drop_would_work?
    return false if row == -1
    return false unless column_id == 'source'
    return false unless pathnames.length == 1
    return false unless /\.rb$/ =~ pathname
    true
  end

  def pathnames
    pasteboard.propertyListForType(NSFilenamesPboardType)
  end

  def pathname; pathnames[0]; end

  private

  def pasteboard; raw_info.draggingPasteboard; end
end
```

fenestra/table-drag-end/app/prefs-window/PrefsTableDraggingInfo.rb

Figure 20.6: PrefsTableDraggingInfo

❹ In the absence of any reason to reject a drop, it should be accepted. In this test, all the defaults hold.

My Solution

My solution is in Figure 20.6. You may also want to refer to the superclass in Figure 20.4, on page 267.

While writing this code, I rediscovered a fact I had learned back on page 238: if you write regexp comparisons with an NSString on the left, you get a RubyCocoa warning. Unlike before, I fixed it by reversing the order of arguments.

It's because of oddities like this that I try to test my Ruby code with NSString. See, for example, how I use to_ns in the should_receive statement within test ❸ in Figure 20.7, on the next page. If the code has to work with Cocoa strings, it should be tested with Cocoa strings.

Download fenestra/table-drag-start/test/prefs-window/prefs-table-dragging-info-tests.rb

```
❶ setup do
    @sut.should_receive(:row, 0).and_return(1).by_default
    @sut.should_receive(:column_id, 0).
          and_return("source".to_ns).
          by_default
    @sut.should_receive(:pathnames, 0).
          and_return(["/path/to/file.rb"].to_ns).
          by_default
  end

❷ should_eventually "reject when nonexistent row" do
    during {
      @sut.drop_would_work?
    }.behold! {
      @sut.should_receive(:row, 0).at_least.once.
            and_return(-1)
    }
    deny { @result }
  end

❸ should_eventually "reject when column_id is not source" do
    during {
      @sut.drop_would_work?
    }.behold! {
      @sut.should_receive(:column_id, 0).at_least.once.
            and_return("display_name".to_ns)
    }
    deny { @result }
  end

  # ...

❹ should_eventually "accept otherwise" do
    assert { @sut.drop_would_work? }
  end
```

fenestra/table-drag-start/test/prefs-window/prefs-table-dragging-info-tests.rb

Figure 20.7: Deciding on a drop

20.6 Try This Yourself: Drag and Drop

Now it's time to put it all together and build PreferencesTableView.

The Strategy

Here are some facts about Fenestra that our solution can use:

- In Chapter 19, *Picking Files with Open Panels*, on page 243, we arranged for PreferencesController to receive HasRubySource notifications. In that chapter, they came from RubyFileChooserController, but there's nothing in PreferencesController that cares about the source of the notification. So, there's no reason I can see not to have our PreferencesTableView post the same notification.
- We've gone to some length to hide NSDraggingInfo's low-level details. So, it seems the dragging protocol methods (like draggingUpdated) should quickly convert the NSDraggingInfo object they receive into a PrefsTableDraggingInfo and then immediately forget about the original object.
- draggingEntered and draggingUpdated evaluate whether what is being dragged can be dropped at the current location. Since they do the same thing, they should delegate their work to a common method.

Putting that all together, I get the skeleton for PreferencesTableView that's shown in Figure 20.8, on the next page.

❶ This method prepares the class to post and receive notifications. All we'll need it for is the post method, originally described in Section 8.3, *Shorthand for Posting Notifications*, on page 103.

❷ All we care about are pasteboards that contain pathnames, but a user could drag anything and try to drop it. This is the code that tells the Cocoa runtime not to bother calling the PreferencesTableView's drag-handling classes unless what's dragged is one or more pathnames.

❸-❹ So that we can quickly forget about NSDraggingInfo, each of the Fenestra dragging methods does nothing but convert its data (using the tailored method defined at ❸) and then call one of three methods (at ❺, ❻, and ❼). Those methods do the real work, and it's those methods that you'll now write.

Download fenestra/table-drag-start/app/prefs-window/PreferencesTableView.rb

```
class PreferencesTableView < OSX::NSTableView

  def awakeFromNib
    # Superclass does not have an awakeFromNib
❶   notifiable_awakeFromNib
❷   registerForDraggedTypes([NSFilenamesPboardType])
  end

❸ def draggingEntered(info); evaluate_location(tailored(info)); end
  def draggingUpdated(info); evaluate_location(tailored(info)); end
  def performDragOperation(info); drop(tailored(info)); end
❹ def draggingExited(info); forget_drag(tailored(info)); end

  testable

❺ def evaluate_location(tailored_info)
    if tailored_info.drop_would_work?
      # ...
    else
      # ...
    end
  end

❻ def forget_drag(tailored_info)
    # ...
  end

❼ def drop(tailored_info)
    # ...
  end

  private

❽ def tailored(raw_info)
    PrefsTableDraggingInfo.alloc.initWithTable_rawInfo(self, raw_info)
  end
end
```

fenestra/table-drag-start/app/prefs-window/PreferencesTableView.rb

Figure 20.8: A starting version of PreferencesTableView

Download fenestra/table-drag-start/test/prefs-window/prefs-table-view-tests.rb

```
context "checking a location where a drop is possible" do
  setup do
❶   @info = rubycocoa_flexmock("drop info")

    @info.should_receive(:drop_would_work?, 0).at_least.once.
          and_return(true)
    @info.should_receive(:row, 0).at_least.once.
          and_return(1)
  end

  should_eventually "select the current row" do
    assert { @sut.numberOfSelectedRows == 0 }
    @sut.fenestra.evaluate_location(@info)
    assert { @sut.numberOfSelectedRows == 1 }
    assert { @sut.selectedRow == 1 }
  end

  should_eventually "signal that a copy is permitted" do
    retval = @sut.fenestra.evaluate_location(@info)
    assert { NSDragOperationCopy == retval }
  end
end
```

fenestra/table-drag-start/test/prefs-window/prefs-table-view-tests.rb

Figure 20.9: What happens when a drop is allowed?

Before the Drop

Refer to Figure 20.9; it shows two tests that describe what should happen when a user has moved the cursor over a Source cell: the current row should be highlighted, and NSDragOperationCopy should be returned.

Note (at ❶) that the @info variable doesn't refer to a real PrefsTableDraggingInfo. Since I believe that class's drop_would_work? and row methods are correct, I don't feel the need to go to the trouble of creating one for this test. Instead, I simulate the values the @sut's evaluate_location method would need. I'll speak more of this decision in Section 20.7, *Does It Work?*, on page 280.

There's another context describing how evaluate_location should work when a drop should not be allowed. There's nothing new or interesting about it, so I won't show it here.

Your mission, should you decide to accept it: make evaluate_location pass the tests in both contexts.

My Solution

Download fenestra/table-drag-end/app/prefs-window/PreferencesTableView.rb

```
def evaluate_location(tailored_info)
  if tailored_info.drop_would_work?
❶   select_row(tailored_info.row)
    NSDragOperationCopy
  else
    deselectAll(self)
    NSDragOperationNone
  end
end
```

The only thing I think of note is at ❶. select_row is one of my shorthand methods from NSTableViewPatches. You can see it in Figure 20.3, on page 266.

Skipping the Drop

Moving the cursor out of the table without ever dropping merits only one test:

Download fenestra/table-drag-start/test/prefs-window/prefs-table-view-tests.rb

```
context "exiting dragging" do
  should_eventually "leave no rows highlighted" do
    @sut.select_row(1)
❶   @sut.fenestra.forget_drag('ignored')
    assert { @sut.numberOfSelectedRows == 0 }
  end
end
```

I can think of no reason why our forget_drag should ever need dragging info, so I deliberately pass it the wrong class of object at ❶: a String instead of a PrefsTableDraggingInfo. You might argue that forget_drag's caller, draggingExited, should simply not hand any argument at all to forget_drag. (See ❹ in Figure 20.8, on page 276, for draggingExited's definition.) I decided against that, though, thinking making all the dragging methods consistent would be less error-prone.[5]

5. When I wrote the original skeleton for PreferencesTableView, my worry about human error (and my "meta mood" at the moment) was enough to have me write code that generated draggingExited and friends in much the way that attr_accessor generates getters and setters. I later decided that was just showing off—and at the cost of clarity—so I threw the code away and wrote the methods by hand.

Now make this test pass.

My Solution

Download fenestra/table-drag-end/app/prefs-window/PreferencesTableView.rb

```
def forget_drag(ignored_info)
  deselectAll(self)
end
```

Dropping

And now for the anticlimax: dropping. Here are two tests:

Download fenestra/table-drag-start/test/prefs-window/prefs-table-view-tests.rb

```
  should_eventually "announce that a row's source should be updated" do
    during {
      @sut.drop(@info)
    }.behold! {
      expected = this_notification(HasRubySource,
                                   @sut,
                                   { :source => @pathname,
                                     :row => 1 })
      watchers_are_notified.once.with(expected)
    }
  end

  should_eventually "return true" do
    assert { @sut.drop(@info) }
  end
end
```

Make them pass.

My Solution

Download fenestra/table-drag-end/app/prefs-window/PreferencesTableView.rb

```
def drop(tailored_info)
  post(HasRubySource, :source => tailored_info.pathname,
                      :row => tailored_info.row)
  true
end
```

Something Else to Try

In writing the code and tests in this section, I assumed that the mouse didn't move between the last time draggingUpdated was called and the time performDragOperation was called. I've spent some time with versions of those two methods that printed out the draggingLocation. I dragged and dropped, trying to find cases where the locations were different. I failed, so I'm comfortable with my assumption.

However, as far as I can tell, Apple doesn't explicitly guarantee that behavior. If you're not as trusting as I am—or if you feel like practicing working with tests—you could write a test for prepareForDragOperation that shows it answering false in cases where draggingUpdated would return NSDragOperationNone. Apple does guarantee that false return value will prevent performDragOperation from being called, so that's the only test you'll need.[6]

Omitting Tests

We have two sets of methods: the "official" ones, such as draggingEntered, and the encapsulated ones, such as evaluate_location. There are no tests that check whether, for example, draggingEntered calls evaluate_location correctly. And there aren't going to be.

Testing, like every other software development activity, involves balancing costs against benefits. In the last century, I thought detailed tests like the ones in this chapter cost way too much to be worthwhile, especially because you had to change them to match a changing implementation. It turns out I was wrong. As computers have gotten faster and people have gained experience, the cost of test writing, running, and maintenance has dropped. Moreover, tests have come to have value beyond just finding bugs. (See Chapter 21, *Epilogue: A Wonderful World of Tests*, on page 282.) So, I, like many others, write many more tests than I dreamed I would ten years ago.

But I still don't see much value in tests to check that draggingEntered really does call evaluate_location or in the elaborate creation of NSDraggingInfo objects just to check that they're in fact converted to PrefsTableDraggingInfo objects. True, looking at the implementations of methods like draggingEntered might not give me enough confidence—I have long experience botching even the simplest methods—but I have another way of gaining it. Read on.

20.7 Does It Work?

The style of testing I've been using in these chapters takes isolation somewhat to an extreme. Most objects under test communicate with

6. If you want to be super scrupulous, note that I can't find any *guarantee* that the NSDraggingInfo passed to performDragOperation is the same as the one given to prepareForDragOperation, so you could write yet another test for the case where they differ. But that way lies madness.

fake objects, not instances of classes they'll work with in the real app. That makes the tests easier to write and—I hope—easier to understand and code to. But it also produces cracks where bugs can slip in.

For example, we've seen cases throughout the book where a Ruby true enters the Objective-C universe and pops out the other side as an integer 1. If we're testing that object at the other end and we "flexmock" the whole Objective-C universe with an object that provides true, we'll have no defense against a nasty surprise when we—or, worse, some user—exercises the real code path.

And we've seen cases where an NSString is importantly different from a Ruby String. It's easy to slip up and test a method with a String, forgetting that it will have to handle an NSString in real life.

One solution is to supplement our automated detailed tests with automated "end-to-end" tests that exercise the whole app or some major chunk of it. Such tests tend to be hard to write and expensive to maintain, so—these days—I lean heavily on careful, disciplined manual testing of changes.

So, carefully try drag and drop. Use the tests as a guide. For example, the fact that there's a test for forget_drag should remind you to drag a file toward a Source cell but then move the mouse cursor out of the table view.

Did you find any bugs?

20.8 What Now?

We're nearly done with this version of Fenestra. I have a few more closing thoughts on the kind of test-driven development we've been doing, but after that, we move to lighter fare: different sorts of minor tweaking to make the app look attractive. For example, perhaps it should have an icon?

And after that, the book is essentially done.

Chapter 21

Epilogue: A Wonderful World of Tests

During all the fun with tables, I've taken some care to present tests as an implementation convenience—which they are. But they're also a design style called *test-driven design* (TDD). I'll explain the basic idea in this chapter.

21.1 Test-Driven Design

Test-driven design, as usually practiced, has four steps:

1. You want to add new behavior to some code. You write a *single* failing test that describes a difference between the code behavior you want and the behavior you have now. The test serves as a small example of one new fact about the code, written from the point of view of a client to that code.
2. You run the test. If it doesn't fail, you fix whatever you did wrong. This step often seems silly until the first time you run the test, the code passes it, and you realize your test either tests nothing or tests the wrong thing.
3. You make the new test pass without breaking any of the tests that preceded it. You try not to write code beyond what the test demands. Good style—like not writing duplicate code—isn't much of a concern.
4. Now that you have a passing test, you look at what you have. Do you have duplicate code? Then group the copies together into a

single method. Does a method seem to do two things? Then break it apart into two methods. Do three methods seem to all be about one topic, something that the rest of the code isn't about? Then perhaps they should go in their own class. Can't remember quite why you picked that variable or method name? Then change it.

The common term for this activity is *refactoring*. When you refactor, you don't change the externally observable behavior of the code; you just make it better internally. If your tests are weak, refactoring is dangerous; if they're strong, it's safe.

You repeat these steps until your code does everything you want (at that moment). Then you move on to some other task.

Design

Test-driven design used to be called test-driven *development* or test-first *programming*. It acquired the *design* label as people noticed that making the tests easy to write, fast to run, and tolerable to maintain forced people to design classes and their interactions differently.

For example, you'll likely find your classes shrink and multiply. Large classes are hard to work with in TDD because they usually require many lines of setup code. With some existing systems, it's practically impossible to create any important object without creating *all* of them. Simple laziness motivates the TDDist to make classes smaller, simpler to create, focused, and independent.

Ironically, TDD forces us to write the kind of code all the slogans told us to: "do one thing and do it well," "classes should be loosely coupled and highly coherent," and so on. We rarely *did*, especially as systems got larger, because doing so meant real and personal pain today for a theoretical benefit that might someday be gotten by somebody (else). TDD brings the cost/benefit equation into the present. Good design means happiness, joy, and ponies *today* because testing gets easy.

Test Doubles

I've been having you use rubycocoa_flexmock without ever giving you terminology for the objects it returns. Gerard Meszaros can fill that gap.[1] All of them are, broadly, *test doubles*. Test doubles are any objects

1. These terms are defined in his *XUnit Test Patterns* [Mes07].

that replace a real object for testing purposes. Meszaros describes four categories of test doubles:

Dummies

A dummy is an object that we need to have but never actually use. In the following code, line ❶ uses a symbol as a dummy argument because a real table would be complicated to build but is never used in the tests.

Download fenestra/table-drag-end/test/prefs-window/prefs-table-dragging-info-tests.rb

```
@raw_info = rubycocoa_flexmock("raw dragging info")
@sut = rubycocoa_flexmock(PrefsTableDraggingInfo) do | klass |
❶   klass.alloc.objc_send(:initWithTable, :unused,
                          :rawInfo, @raw_info)
```

Stubs

A stub gives the same results as the real object would but can work only in your test (because it couldn't handle everything the real world might throw at it). In the following, I've "stubbed out" the clickedColumn and clickedRow methods:

Download fenestra/table-drag-end/test/prefs-window/prefs-change-source-tests.rb

```
@table.should_receive(:clickedColumn).and_return(col)
@table.should_receive(:clickedRow).and_return(row)
```

Mocks

A mock is programmed with *expectations* about how it should be called. Here's one of our examples:

Download fenestra/table-drag-end/test/prefs-window/prefs-change-source-tests.rb

```
@table.should_receive(:editColumn_row_withEvent_select, 4).once.
       with(@display_name_column_index, 1, any, 1)
```

The method has to be called with a certain set of arguments. Moreover, it has to be called exactly once. If either of those expectations isn't satisfied, the test fails.

Mocks can return values, but we haven't seen an example that both returns values and checks its arguments.

Fakes

A fake does the same thing as the real object but in a more convenient way. For example, you might replace an instance of Oracle or MySQL with some in-memory database. I didn't use any fakes in this book.

21.2 To Learn More

Because I've treated test-driven design as nothing more than a convenience, I recommend you next read the book that concentrates on TDD as *design*: Kent Beck's *Test-Driven Design: By Example* [Bec02]. After that, Ron Jeffries' *Extreme Programming Adventures in C#* lets you watch over the shoulder of someone to whom TDD is second nature. Like this book, it builds up a single larger application rather than showing small examples.

As with TDD as a whole, I used mock objects just as a way to save effort. There's a school of practitioners who use mock objects heavily as a design tool. That leads to a somewhat different workflow and emphasis than the previous two books show. There is no published book that covers this style, but there will soon be: Freeman and Pryce's *Growing Object-Oriented Software, Guided by Tests*. As of March 02009,[2] you can find drafts at http://mockobjects.com/book. That site also has useful papers and essays on design with mock objects.

2. Sic. I like to use the Long Now Foundation's dating scheme: "The Long Now Foundation uses five-digit dates; the extra zero is to solve the deca-millennnium bug that will come into effect in about 8,000 years." The Long Now Foundation "was established in 01996 to creatively foster long-term thinking and responsibility in the framework of the next 10,000 years." See http://www.longnow.org.

Part VI

Wrapping Up

Chapter 22

Fit and Finish

In this chapter, we'll tinker with Fenestra to make it look and behave more like a respectable Mac app. We'll add expected behaviors like responding nicely to Tab, restoring window positions when the app restarts, and resizing window contents along with the window frame. And we'll remove jarring visual touches like the pervasive use of "New Application" instead of "Fenestra," an About window that says nothing about Fenestra, and buttons that slightly overlap.

22.1 Saving the Window Position Until the Next Launch

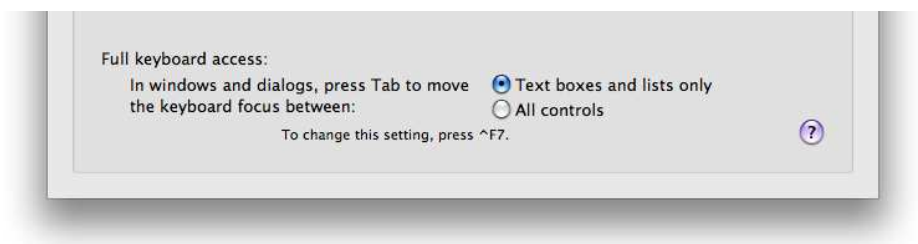
In Section 11.5, *Try This Yourself: A Sticky Window*, on page 145, we wrote code that explicitly saved the window position as a user preference. Since practically every app wants to do that, there's an easier way: simply set the window's *frame name* in its Attributes inspector:

The named window prompts Cocoa to save the window position and size in user preferences. (The frame name is used as the key for that data.)

Make that change to your latest version of Fenestra; then try it.

22.2 Tab Behavior

As you'll see shortly, Fenestra's default response to you pressing a sequence of Tab keys is clumsy. In this section, you'll improve it. However, since Fenestra's user interface doesn't have many controls, first set your system preferences so that Tab can take you to all of them. You do that with the Keyboard & Mouse panel inside System Preferences. On the Keyboard Shortcuts tab, you'll see the following:

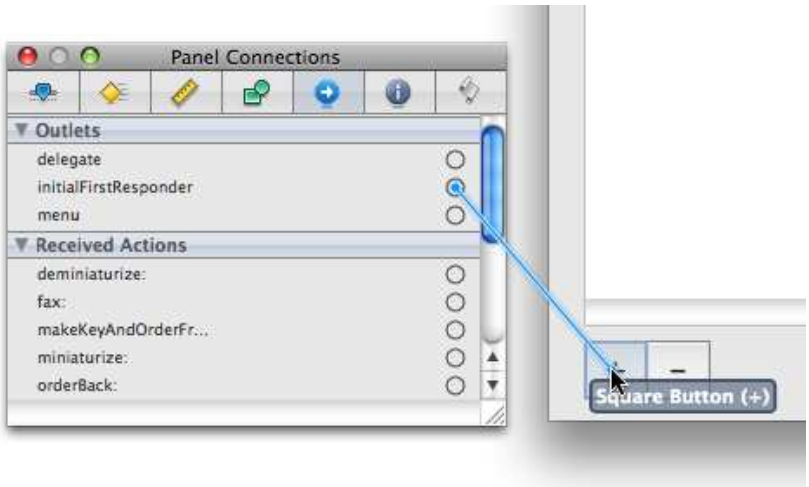


Change it to "All controls."

Now launch Fenestra, and open the preferences panel. Start pressing Tab. I expect that you'll see each field in the first row of the table become enabled for editing and then see the add and remove buttons highlighted with a light blue focus ring. The focus ring means that you can press the button by pressing the spacebar. Here's an example—even in black and white, you should be able to tell that there's a fuzzy ring around the remove button:

Let's pretend that we usually want Tab to cycle between the two buttons. It shouldn't ever take you to a new table cell unless you started from one (by putting the cursor there with a click). We can do that by taking control of the *key view loop*.

We start to define the key view loop at the window. It controls which of its views first gets focus.[1] Open the Connections inspector for the preferences panel. Drag to create a connection between the initialFirstResponder outlet and the add button:



Go now to the Connections inspector for the add button. There's an outlet named nextKeyView there. Make a connection between it and the remove button.

To complete the loop, make a connection between the remove button's nextKeyView and the add button.

Now see how tabs work, either by using IB's interface simulator or by building the real app.

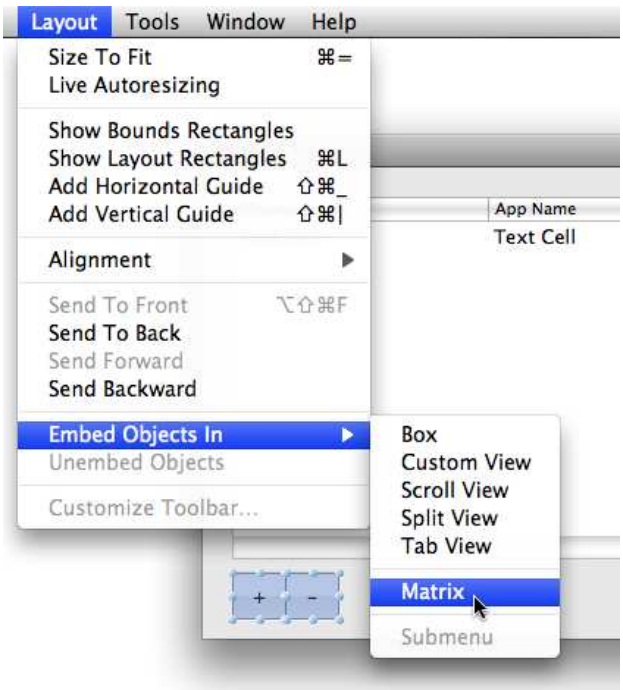
22.3 Using NSMatrix to Organize Buttons

You may have noticed before now that the add and remove buttons slightly overlap. Giving them focus rings makes that painfully obvious. Compare these two images:

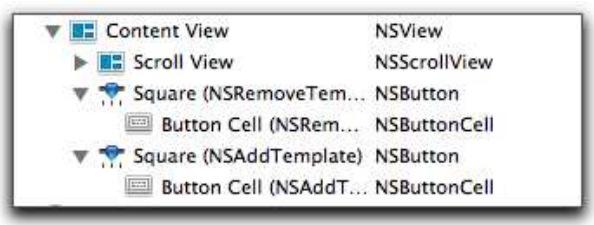
1. Remember that every control is a view.

What's happening here is that the remove button is ordered on top of the add button so that its nonfocus image obscures part of the add button's focus ring.[2]

We need to put the two buttons at the same level. We can do that by putting them inside a *matrix*, an instance of the NSMatrix class. Do that by selecting both of the buttons and choosing the menu item Layout > Embed Objects In > Matrix, as shown here:

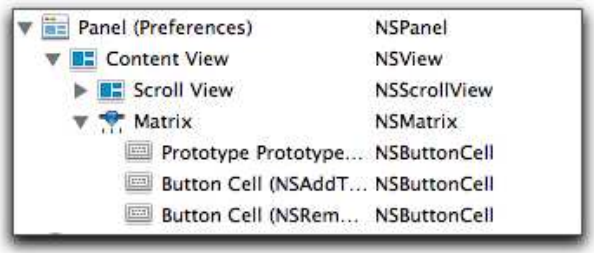


You'll see little visible change. However, you've just changed this view structure:



2. If you created the buttons in a different order than I did, you may see the add button on top of the remove button.

. . . into this one:



You've discarded two buttons and replaced them with one matrix. The cells behind the buttons have been retained in the matrix. Recall, from Section 17.1, *Cells*, on page 230, that controls like buttons delegate most of their work to cells. What work they don't delegate is the same for buttons and for fixed-size matrixes, so the only change a user could observe would be a tidier focus ring.

You can consider an NSMatrix very roughly like an NSTableView. An NSMatrix object has rows and columns that contain cells. New rows and columns can be added. (In our case, the new cells would be copies of the Prototype shown in the previous snapshot.) An NSMatrix object supports editing, scrolling, and sorting. However, they don't have the added structure given by NSTableColumn and NSTableHeaderView.

NSMatrix objects are often used to group cells that interact, such as radio buttons. In Section 16.3, *Interdependent Favorite Values*, on page 222, you wrote code that observed the result of typing "yes" in one Favorite cell and made the current favorite no longer the favorite. In Chapter 17, *Buttons in Tables*, on page 230, the text fields in the Favorite column were changed to radio buttons. That made your code implement typical radio button behavior: select one, and another gets deselected. An NSMatrix containing buttons can do that kind of work for you if you set its mode to NSRadioModeMatrix.

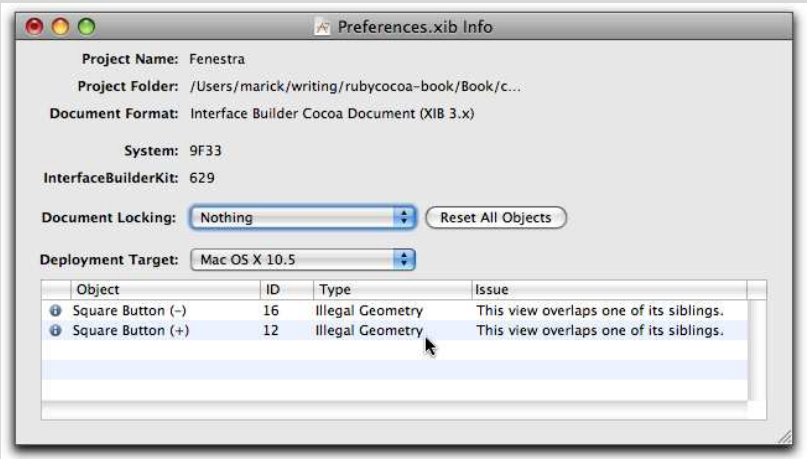
For more, see the NSMatrix API documentation and Apple's *Matrix Programming Guide for Cocoa* [App08p].

Tab Order Again

Grouping the buttons destroyed the destination for the window's initialFirstResponder. Restore it by making a connection between the window and the matrix. You don't need to make a nextKeyView connection

Noticing Interface Builder Warnings

At any time, you can look at a nib's Document Info window (under the Window menu). You'll see a list of warnings. In the case of the overlapping buttons (Section 22.3, *Using NSMatrix to Organize Buttons*, on page 289), you'd see this:



between the two matrix cells: by default, a matrix makes Tab move through its cells in order, starting from the upper left.[3]

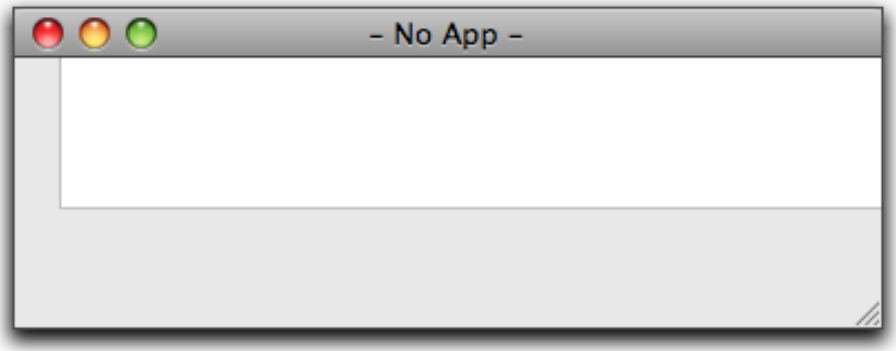
Run Fenestra. You should see the add button selected. Press the spacebar. As before, you'll get a new table row and be editing the first cell. Edit the row, tabbing through the rest of the cells. When you tab out of the last cell, you'll notice that the entire table will be selected. What happens if you press the spacebar now?

Is that what you want? Or do you want to be able to keep pressing the spacebar to add more rows? If so, make the NSTableView's nextKeyView the matrix. Is the new behavior more to your liking? What happens when you click a row to select it and then click the remove button to remove it? Do you like that behavior? Is there behavior you'd like better?

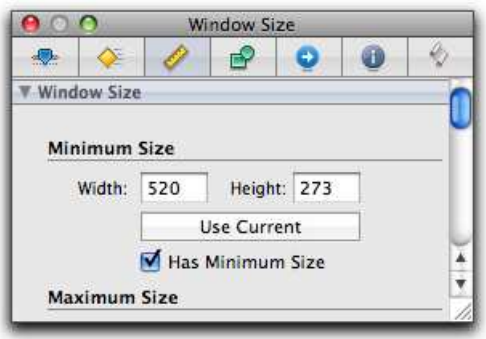
3. Indeed, you can't set the nextKeyView on the cells. nextKeyView is a property of an NSView. Controls are NSView objects, but cells are not.

22.4 Sizing

Run Fenestra. Resize the main window (not the preference panel) to make it smaller. The app behaves badly:

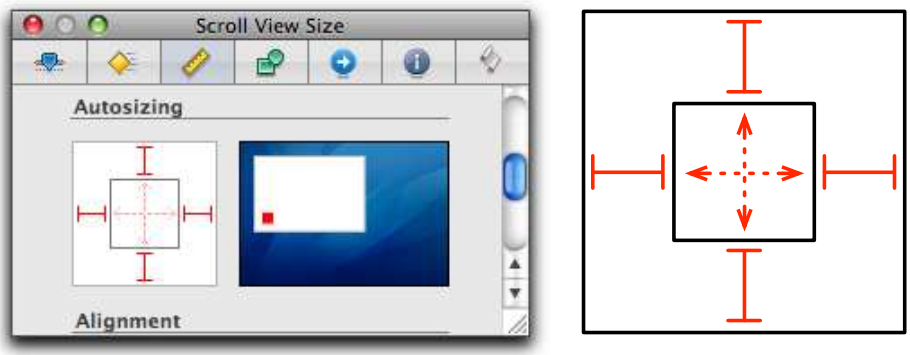


Let's fix that by giving the window a minimum size. Open MainMenu.nib. With the window selected, go to the Size inspector. (It's the third tab.) Select the Has Minimum Size checkbox, and click the Use Current button:



Run the IB simulator. You should no longer be able to shrink the window. What if you grow it, though?

That's not so good, either. Let's first make the text view grow along with its containing window. Select the text view. The Size inspector will show you the following:



The key part is the little box diagram, which I've reproduced in a larger size on the right.

Within the diagram, the inner box represents the inspector's view (in our case, the text view). The outer box is the containing view (in our case, the content view that covers the entire window). The outer solid lines that look reminiscent of I-beams mean that the window system should try to keep constant the distance between that edge of the text view and that edge of the containing view. The inner dashed arrows mean that the text view is not allowed to grow.

When you grow the window, the text view can't both stay the same size and stay the same distance from the edges of the content view. Staying the same size takes precedence.

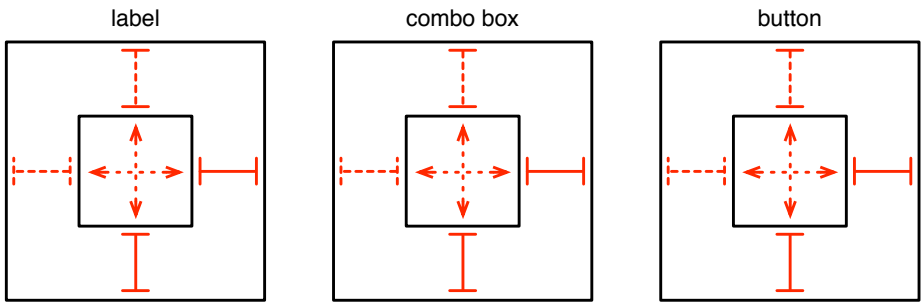
In that case, the best the window system can do with the distance between edges is keep it the same for two pairs.

To make the text view grow to match the window, click the two interior dashed lines to make them solid. Note the behavior of the inspector's useful-yet-annoying animation.

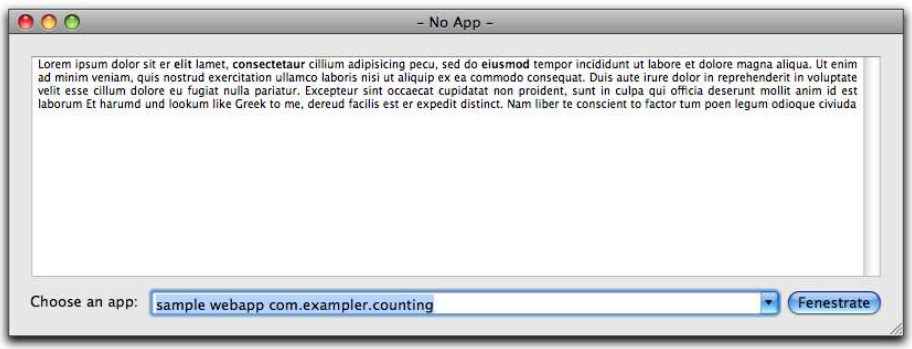
Simulate the interface, grow the window, and behold:

The text view behaves as we desire. The next step is to fix the combo box, label, and button. One thing we might do is have them "stick" to the bottom-right corner and move as the user grows the window. Here's a snapshot:

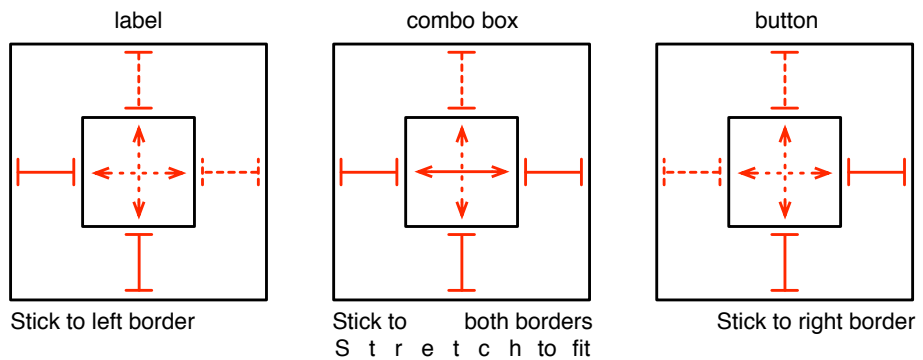
That can be done by setting the controls so that they remain the same distance from the bottom-right edge of the window's content view. That is shown here. (Since the settings are identical for all three, you can select them and edit them as a group.)



However, one reason I might want to grow the window is because a particular display name is too wide to fit in the combo box. If so, I'd prefer the combo box to grow with the window. Like this:

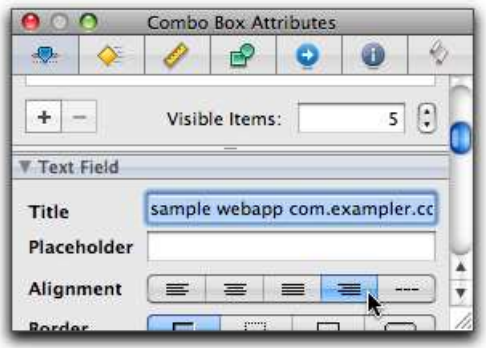


That can be accomplished by setting the three controls as follows:



The key trick is turning the horizontal dotted line inside the combo box's view into a solid line by clicking it. That means the view is allowed to grow horizontally.

But—never satisfied, am I?—now I don't like the long distance between the text in the combo box and the button. It's an enormous imposition to have to drag the cursor all that way just to click. I'd rather see the combo box's text right-aligned. That's easy to do with the Attributes inspector:



22.5 Cleaning Up the Menu Bar

The default menu bar contains menu items irrelevant to this app. You can delete them by selecting them and pressing `Delete`.

Deleting an entire menu is a little more complicated. The first time you click it, it shows the items it contains. If you press `Delete`, then it won't delete the menu. One or two more clicks will make the items vanish, keep the menu selected, and let you delete it.

I recommend you experiment with the different menu items in order to see which ones already work. Delete the ones that don't.

The second menu bar task is to change all the instances of "New Application" to the real application name. You can find them in (and under) the New Application menu. There's also one under the Help menu.

22.6 The About Window

The application menu has an About item. The window it pops up also needs changing.

By default, it will look something like this:

You may have already noticed the annoying *__MyCompanyName__* when creating source files in Xcode, but it's doubly annoying here. To change it, edit English.lproj/InfoPlist.strings.[4] Change CFBundleGetInfoString and NSHumanReadableCopyright.

You probably do not want to make this change for every new project. I see no way to change the default for all Xcode projects, but you can set it through the command line:

```
2116$ defaults write com.apple.Xcode PBXCustomTemplateMacroDefinitions ↩
'{ORGANIZATIONNAME = "Exampler Consulting";}'
```

You'll also want to change the version numbers. They come from two entries in Info.plist: CFBundleShortVersionString and CFBundleVersion. The former is the "marketing version" and is typically manually changed once per public release. The latter is for internal use and is often changed automatically (such as upon each successful build or each version control check-in).

You will undoubtedly want a special icon for your app. You can use /Developer/Applications/Utilities/Icon Composer to create one. Simply create a TIFF-format file, and drop it onto the largest of Icon Composer's Images panes. Hint: Icon Composer won't scale images up to fill the space, but it will scale them down, so it's easiest to drag in a giant image.

4. English.lproj contains localizations for the English language. You can create other folders for other languages. See *Internationalization Programming Topics* [App08l].

For your convenience, I've created Fenestra.icns and put it in the fenestra directory. Installing the icon is a two-step procedure:

1. You have to tell Xcode about the file. Either drag it onto the project window's Resources group or select that group and use Project > Add to Project.[5]

   After you drag the file or pick it with a file chooser, Xcode will pop up a dialog box you've seen before (for example, on page 112). Unlike in those earlier cases, Fenestra.icns isn't already in the project folder, so make sure you select the "Copy items into destination group's folder" option.

2. The CFBundleIconFile entry in Info.plist tells the app which icon file to use. It starts out looking like this:

   ```
   <key>CFBundleIconFile</key>
   <string></string>
   ```

   Change it to this:

   ```
   <key>CFBundleIconFile</key>
   <string>Fenestra</string>
   ```

Gaze in wide wonder:

22.7 Changing the Application's Name

In Section 11.1, *The User Preferences System*, on page 128, you saw that Fenestra's preferences are stored in ~/Library/Preferences/com.apple.rubycocoa.FenestraApp.plist. That's not the best name, but it's easy to change.

5. It doesn't actually matter where the icon ends up, but Resources is the typical location.

Edit Info.plist to change this:

```
<key>CFBundleIdentifier</key>
<string>com.apple.rubycocoa.FenestraApp</string>
```

. . . to this:

```
<key>CFBundleIdentifier</key>
<string>com.exampler.Fenestra</string>
```

Chapter 23

Adding Help

An application without help can make users sad. Apple's *Help Programming Guide* [App08a] will help you provide help. I wrote this chapter because the guide seems to assume you know things about help that I didn't when I started out.

Although most of the chapter is about creating the *help books* that users reach through the Help index, I also describe how to add tooltips to window controls.

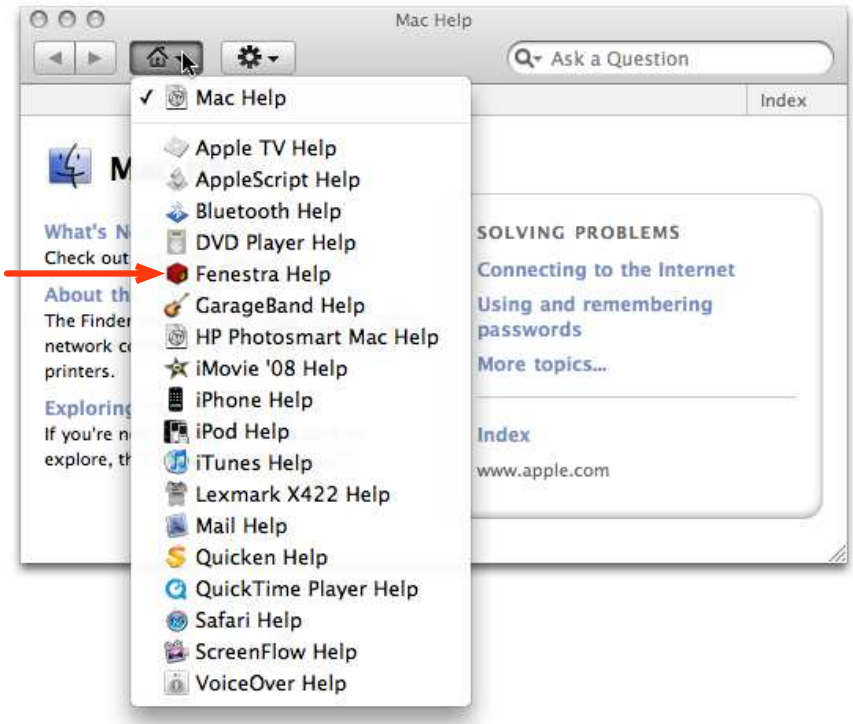
If, like me, you've never paid attention to the structure of help books, I recommend you browse through Fenestra's now. It has a title page, two content pages, and five index pages. Use the version in fenestra/fit-and-finish.

23.1 Help Book Basics

Most help books live within an app's bundle. You can also have the help viewer fetch pages across the network from your server, but I won't cover that in this chapter.

When you ask an app for help, the help book is cached inside ~/Library/Caches/com.apple.help. Because of that, you can read an app's help book even when the app isn't running.

That's done through this menu in the help viewer:



Help pages can contain hyperlinks. Many of those are not direct links but rather searches, which can make help seem unresponsive.

23.2 Creating a Help Book

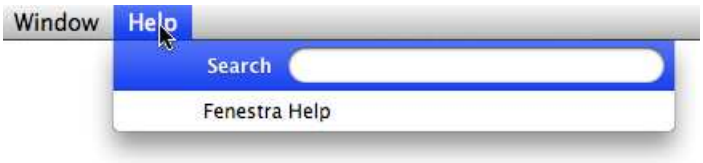
Help books are written in XHTML. With the exception of some special-purpose <*meta*> tags and <*a*> *hrefs*, it looks like any other CSS-heavy HTML.

Since Apple's help books contain files with descriptive names like 10028.html, I infer they're generated by some tool. As far as I know, you can't use it. The Omni Group has a free package, Helpify,[1] that converts OmniOutliner files into help books. I built Fenestra's help book by hand.

1. http://blog.omnigroup.com/2008/10/02/helpify-the-omni-help-emitter/

My help book is in fit-and-finish/English.lproj/FenestraHelp. It contains these files and folders:

FenestraHelp.html
: This is the starting help page. You get it when you pick Fenestra Help from this menu:



FenestraHelp.helpindex
: This file is generated (see Section 23.4, *Indexing Page Content*, on page 310). It contains information used in responding to user searches. It has nothing to do with index pages.

sty and gfx
: The sty folder contains Apple's CSS stylesheets for help pages. The stylesheets make use of graphics files in gfx.

  Important: One of the files in sty describes the results of a search. It contains the hard-coded name of a help book. If you reuse these stylesheets, search for all instances of *Fenestra Help* in sty/genlist.html, and change the app name to match your app. If you don't, the Home and Index links on your app's search results page will try to take the user to Fenestra's help book.

images, index-pages, movies, pages
: You can put content anywhere you like within the root help folder. Apple uses short, obscure names for all the help book folders. I like longer names, so I ignored Apple's choices and gave my content folders these four names. I kept sty and gfx because there are hard-coded references to them all over the place.

23.3 Editing Pages

There's no documentation for Apple's CSS that I could find. So, you can learn it in the time-honored way: by copying and tweaking someone else's HTML. That's what I did to learn most of what follows.

Figure 23.1: A help book's title page

A Title Page

The title page of Fenestra's help book is shown in Figure 23.1. I copied it from Mail's help.

Here's the head of its source:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/FenestraHelp.html

```
<head>
  <title>Fenestra Help</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
❶ <meta name="AppleTitle" content="Fenestra Help" />
❷ <meta name="AppleIcon" content="FenestraHelp/images/fenestra16x16.png" />
❸ <meta name="robots" content="anchors" />
  <link href="sty/access_4box.css" rel="stylesheet"
        type="text/css" media="all"/>
</head>
```

❶ *AppleTitle* identifies this as the first page of the help book. It's also the name that appears in the drop-down list of help books shown in Section 23.1, *Help Book Basics*, on page 301.

❷ This small image is displayed alongside the help book's name in the drop-down list of help books. It should be a 16x16 PNG image.

❸ Normally, all the text on the page is indexed for searching. A page like this one isn't of much use as a search result, so this tag tells the indexer to look only at descriptions of links to other pages.

You can see the "navbox" in Figure 23.1, on the preceding page—it's the strip across the top of the window. The navbox contains a link to the main index. In all pages but this one, it also contains a link back to this page. Here's the source for the navbox:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/FenestraHelp.html

```
<div id="navbox">
❶  <a name="access"/>
  <div id="navleftbox">
  </div>
  <div id="navrightbox">
    <a class="navlink_right"
❷      href="help:anchor='xall' bookID='Fenestra Help'">
      Index
    </a>
  </div>
</div>
```

❶ This is the name of the page for linking or searching purposes. Apple seems to always use *access* for the title page's name, so I am too.

❷ We could use a direct link to the index page, but that would make the help viewer too responsive.[2] *help:anchor* uses the search index from the help book named by the *BookID* to find the page named by the *anchor*. In this case, that's the Fenestra page named *xall* (which is the name of the main index).

The application's icon conventionally appears in the header of all pages. That's accomplished at ❶ within this HTML:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/FenestraHelp.html

```
<div id="headerbox">
  <div id="iconbox">
    <img id="iconimg" src="images/fenestra32x32.tiff"
❶        alt="Fenestra Help Icon" height="32" width="32"/>
  </div>
  <div id="pagetitlebox">
    <h1>Fenestra Help</h1>
  </div>
</div>
```

2. Ha ha! Only serious!

Figure 23.2: A help book page

The image should be 32x32. An icon of that size can be dragged out of the application's .icns file onto disk. So can the 16x16 icon, but it has to be converted from TIFF format to PNG format.

The rest of the HTML is the usual unholy mess of <*div*> tags that is modern web page implementation.

A Content Page

An ordinary help book page is shown in Figure 23.2.

There's not much new in the HTML. The metadata, shown next, has no instructions for robots, so all the text on the page will be indexed.

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/pages/movie.html

```
<head>
  <title>Fenestra in Action</title>
❶   <meta name="description" content="A movie showing what Fenestra can do"/>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <link href="../sty/access_4box.css" rel="stylesheet"
        type="text/css" media="all"/>
</head>
```

The *description* <*meta*> tag (at ❶) gives text that appears in a tooltip when you hover over a search result that links to this page. Like this:

The navbox section of the HTML also has some differences:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/pages/movie.html

```
<div id="navbox">
❶   <a name="movie"/>
❷   <a name="almainwindow"/>
❸   <a name="alwindows"/>
❹   <a name="almusic"/>
```

The page has been given four names. The first will be used in a *help:anchor* search. The latter three are used in a *help:topic_list* search. You can see how that's done in Section 23.3, *Index Pages*, on the following page.

After the names comes more unexciting HTML, one snippet of which is this:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/pages/movie.html

```
<embed src="../movies/fenestra.mov" width="584" height="314"
       autoplay="false"/>
```

Figure 23.3: A help book index page

That embeds a movie. If the <*embed*> tag didn't give a *width* and *height*, the movie wouldn't be scaled to fit. Instead, it would be cropped (and useless). So, don't forget those attributes when you embed movies. The same goes for images.

For movies, the *height* of the embedding should leave room for a 16-pixel control strip on the bottom.

Index Pages

A help book index page is shown in Figure 23.3.

This particular page shows all the index entries. Other pages show only entries beginning with a particular letter. A single line in the output is generated with this HTML:

Download fenestra/fit-and-finish/English.lproj/FenestraHelp/index-pages/xall.html

```
<tr>
  <td>
    <a href="help:topic_list=alwindows
                bookID='Fenestra Help'
                template=sty/genlist.html
                stylesheet=sty/genlist_style.css
                Other=Windows">Windows</a>
  </td>
  <td></td>
</tr>
```

Figure 23.4: A cross-reference page

When clicked, that "link" generates a page containing links to all pages named with *alwindows*. Such a page is shown in Figure 23.4.

The *Other* attribute puzzled me for a while. It's the title and level-one header for the generated page. As such, it should almost always be the same as the text content of the <a> tag (*Windows* in this case). But you can change it to, oh, *Wombat* if you like.

23.4 Hooking a Help Book into an App

Making a help book available requires a little bookkeeping work.

Xcode

You should add the help book to Xcode in the same way we added the third-party folder in Section 9.1, *Preparing a Directory*, on page 111. Otherwise, the help book won't be copied into the application bundle at build time.

Info.plist

Info.plist needs two entries about the help book, as shown here:

```
<key>CFBundleHelpBookFolder</key>
<string>FenestraHelp</string>
<key>CFBundleHelpBookName</key>
<string>Fenestra Help</string>
```

The first tells the app where to find the help book the first time it's used. The second gives the name that's used in searches and the drop-down list of help books.

Indexing Page Content

Help book searching relies on an index generated by the help indexer app. You pick a folder, and it indexes it. The interface is somewhat clumsy. In particular, it pops up this message after indexing:

There's actually no reason to quit. You can ignore the pop-up and leave the app running. Each time you want to reindex, you can just press Command-I to get the starting dialog box. After that, the key sequence spacebar-Return-Return will reindex your help files.[3]

The help indexer's help page describes how to run it from the command line. That allows you to make reindexing an automatic part of the build.

3. In order to use the spacebar to press a button, you must have set your Keyboard and Mouse system preferences as described in Section 22.2, *Tab Behavior*, on page 288.

23.5 A Workflow for Creating Help Book Pages

Here are some of the facts that drive my workflow:

- You can view help pages in Safari. Provided the tags for images and videos contain the *length* and *height* attributes, I can't tell the difference between its rendering and the help viewer's.
- When Safari follows a link that uses the *help:* protocol, it launches the help viewer.
- If the app doesn't provide an index, the help viewer will build one for you. Errors in your HTML are reported less usefully, though, and warnings aren't reported at all.
- After you've provided the index the first time, you need to keep reindexing pages after you change them. If you don't, the help system won't notice the changes.

And here's my workflow:

- While working on a page, I proofread it in Safari. That's much faster than starting the help viewer and navigating to the page.
- When it comes time to check a page's links, I rebuild the help index and then rebuild and launch the app. I don't go to help through the app, though. Instead, I use Safari to click the links.

Early in my exploration of help pages, I managed to get the help viewer's notion of my help book completely out of sync with my source. I reverted to a blank slate with these two commands:

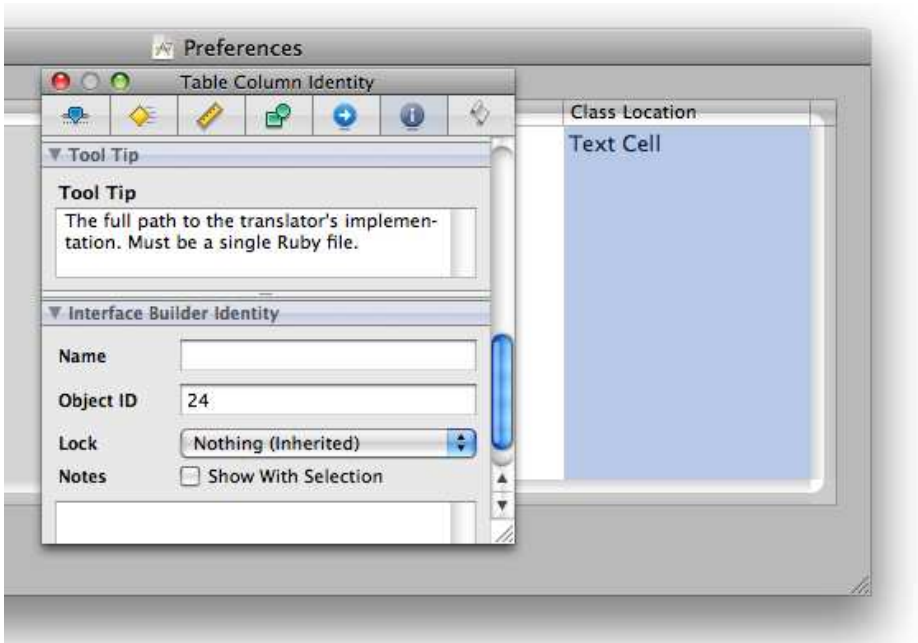
```
rm -rf ~/Library/Preferences/com.apple.help*
rm -rf ~/Library/Caches/com.apple.help*
```

23.6 Tooltips

The behavior of Fenestra's main window is intuitively obvious.[4] But it's conceivable—*barely* conceivable—that some user might need help understanding what to put in the different columns of the preference pane. To help them, I'll add a tooltip to each column.[5]

4. To me.
5. The tooltip appears when you hover over the column heading. I can't see a way to make it appear when you hover over a column cell.

Tooltips are defined in IB's Identity inspector:

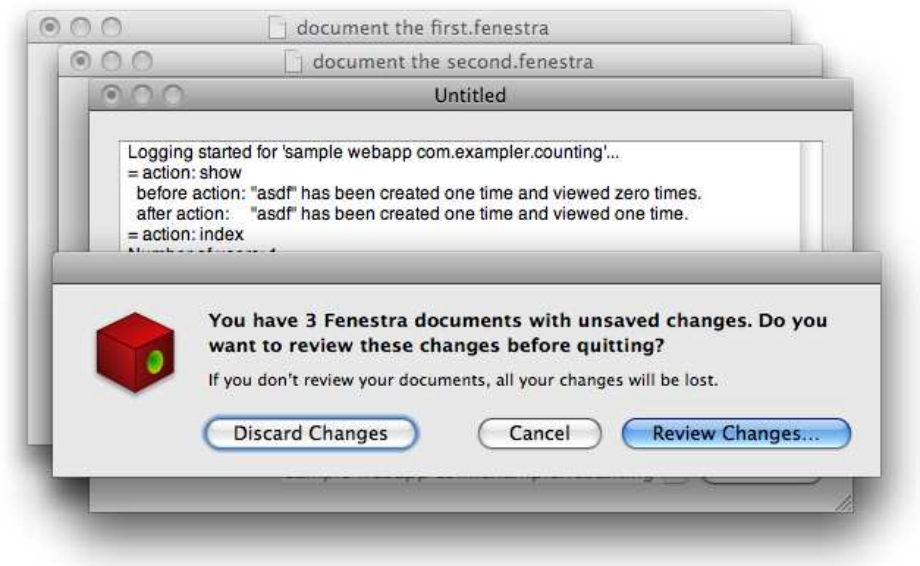


Chapter 24

Document-Based Applications

Most Mac applications are *document-based*. You create a new document with Command-N, edit it, save it with Command-S, reedit it later with Command-O, and so on. Should Fenestra be document-based? I can't see why. The most I'd want is a button to save a log to a file.

Still, I'd be doing you a disservice if I didn't show you how document-based applications work, so I've produced a version of Fenestra that's document-based. It looks like this:



Spike Solutions

The version of Fenestra in this chapter is what some people call a *spike solution*. A spike solution, or *spike*, is a coding exercise done in response to a problem. You "spike it" for long enough to know how the problem should be solved; then you throw it away.

Spikes aren't suitable for real use, and neither is this version of Fenestra. Document creating, saving, and reopening all work. Incoming notifications are translated and added to open documents. But I haven't bothered with incidentals that would make this a *useful* document-based app. For example, each notification is added to every open window.

For my spike, I had to manually add the mechanisms behind document-based applications to Fenestra. That was a useful exercise since it forced me to discover every piece of the puzzle. But I could have avoided some of that work when I created Fenestra, way back on page 38. There, I chose to make it a Cocoa-Ruby Application. Had I chosen Cocoa-Ruby Document-based Application, a different set of empty nib and template files would have been created.

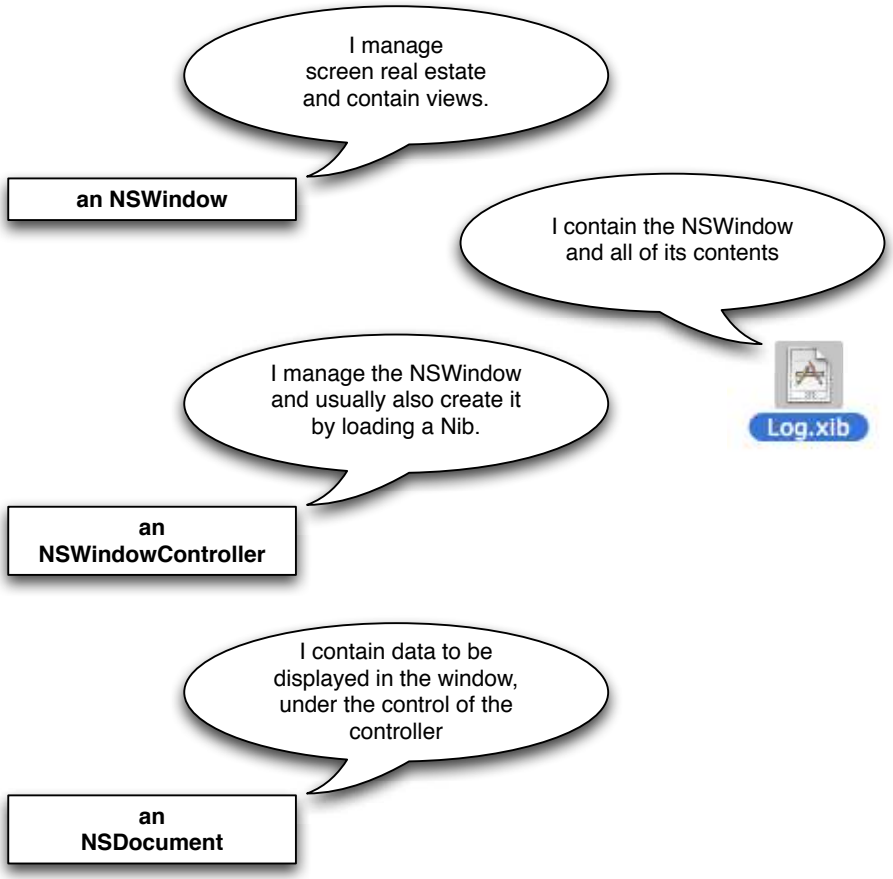
Notice that it has some of the properties you expect from a document-based application: the title bars are filenames or "Untitled," Fenestra documents have their own extension, each new window was created offset from the next, and I got a warning pop-up when I tried to quit Fenestra without saving changes.

24.1 The Major Players

Most of the objects involved in a document-based application play familiar roles, as shown in the following image. There's a window containing views, a controller that manages it, and a source for the data the window displays.[1] The only new twist is that the window and its views

1. I wanted this to be the only Cocoa book in history not to name this three-object structure, but that would probably be irresponsible. Without that password, you'd never

are most likely loaded from their own nib file, one separate from the MainMenu.nib that describes the application as a whole. In a document-based application, MainMenu.nib is mostly about the menu bar.



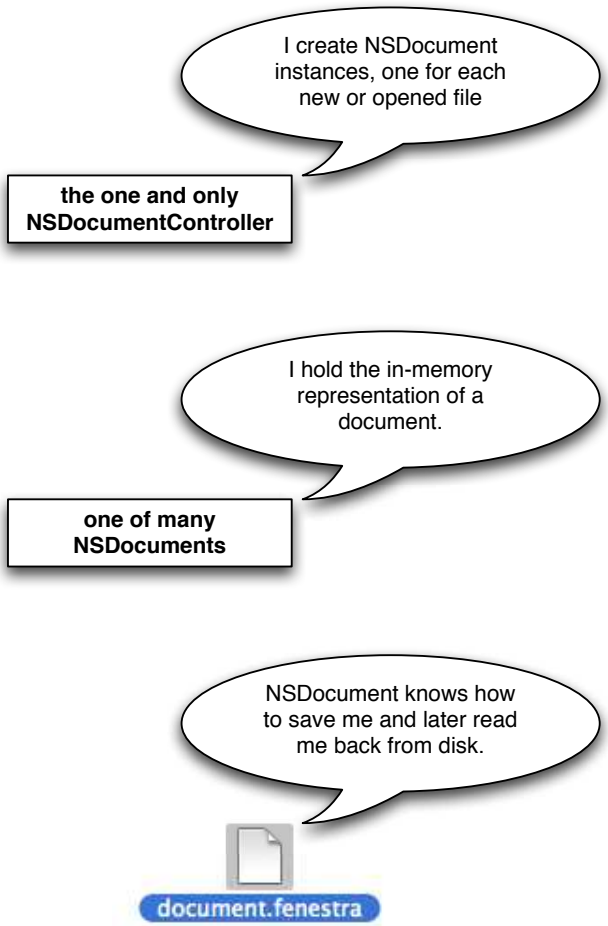
In simpler programs, NSDocument and NSWindowController can be combined into one object, but I'll keep them separate in Fenestra.

Because a document-based app may have many documents open, there is another object with the responsibility of handling them, NSDocumentController.

---

be granted entry into the Cocoa Club. So, OK: this structure is called the *model-view-controller pattern*, or *MVC*.

The following picture shows how it fits in:



Note that the NSDocument has the responsibility of saving itself to disk and of converting the on-disk document format into an in-memory format.

24.2 The Responder Chain

Find a document-based application (the version of Fenestra located in document-based-spike will do fine) and open its MainMenu.nib. Open one of the File menu's items—New, say. Look at the item's target in the Connections inspector.

You'll see this:

Since it's the document controller that creates new documents, why is the menu item's target the First Responder pseudo-object? Shouldn't it be the document controller?

The first responder was briefly explained in Section 12.5, *IB's First Responder Pseudo-Object*, on page 159, but this is a good point at which to say more. "First Responder" is just a fancy way of saying nil: there is no target object. Rather than throwing a null-pointer exception, the Cocoa runtime takes the nilness as a signal to search for some object that'll accept the action.

Let's suppose that we clicked New while a Fenestra window's combo box had focus. The runtime has to find an object that accepts newDocument:. A picture of a search, along the combo box's *responder chain*, is shown in Figure 24.1, on the next page. Here's its explanation in words:

1. Does the combo box handle newDocument? No.
2. The window has a content view that contains the combo box. It is the next element in the responder chain. Does the content view handle newDocument? No.
3. Perhaps the window itself handles it? No.
4. Does the window have a delegate? Perhaps that handles it. No.
5. There's an NSWindowController that mediates between the NSDocument and the Fenestra window that's displaying it. Perhaps *that* handles it? No.
6. What about the NSDocument itself? No.
7. What about NSApplication? Does NSApplication handle it? No.
8. The NSApplication's delegate? No.
9. That leaves only the DocumentController. You don't suppose...? It *does* handle newDocument? Success!

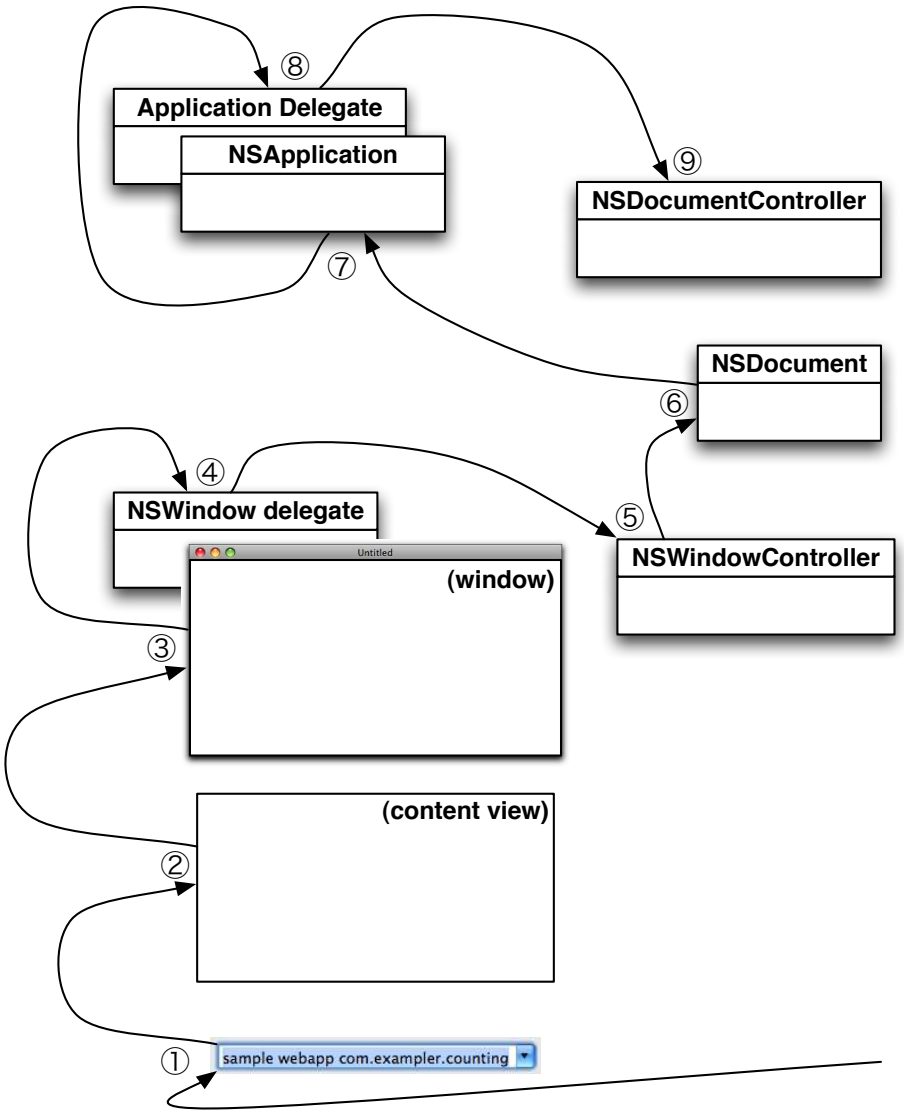


Figure 24.1: The responder chain

If the action were, instead, saveDocument:, the search would stop at the NSDocument, the first object in the chain that defines that method.

It's possible for one window to be handling keypresses while another handles clicks. In that case, the search is more complicated. See Apple's *Cocoa Event-Handling Guide* [App08h] for all the details. The good news is that searching usually just works: the right actions get called on the right objects without you having to worry about it.

24.3 Creating a New Document

At 6:34 p.m. on February 8, 2009, I launched Fenestra and immediately pressed Command-N. A new Fenestra window appeared. In this section, I'll devote hundreds and hundreds of words to what took about a second in real life. I'll also explain the setup work required to make that second's work a pleasant confirmation rather than a nasty surprise.

One word of explanation first, though: most document-based applications create an untitled document when they're launched. I've turned that off in Fenestra because explaining those steps wouldn't add anything to the steps required to respond to Command-N. (They're just a subset.)

Turning off the default behavior is done by adding this method to the application delegate:

Download fenestra/document-based-spike/app/main-menu-window/AppDelegate.rb

```
def applicationShouldOpenUntitledFile(sender)
  false
end
```

That said, you can see what appeared in Fenestra's debug log at around 6:34:02 p.m. in Figure 24.2, on the following page.

What Defines a Document-Based Application?

Unless the application is document-based, the New menu item (if it exists) is inactive, and Command-N does nothing. A document-based application is one that has a CFBundleDocumentTypes key in its Info.plist. The key's value is an array describing the different kinds of documents the app will handle.

```
DocumentController awakes from nib.
DocumentController asked to create a document.
Document springs into life!
Document will now create its controller.
LogWindowController springs into life!
LogWindowController will load the nib named 'Log'.
LogWindowController has been given a pointer to Document.
Document has been asked to show its windows.
LogWindowController has been asked to show its window.
  (Is the Nib actually loaded yet? false)
LogWindowController is told the Nib is loaded.
LogWindowController now has an outlet to OSX::NSWindow.
LogWindowController stuffs '' into text view.
```

Figure 24.2: Creating a new document

Fenestra handles one, and the important parts of its description are these:

Download fenestra/document-based-spike/Info.plist

```
<key>NSDocumentClass</key>
<string>Document</string>
<key>CFBundleTypeExtensions</key>
<array>
  <string>fenestra</string>
</array>
```

There's one kind of document, handled by an NSDocument subclass I'll name Document. Such a document is stored in files ending in .fenestra.

Creating a Custom Document Controller

There's always one instance of NSDocumentController, which can be gotten with this statement:

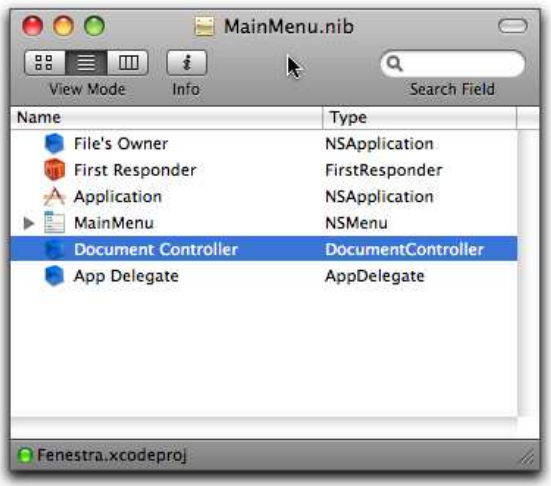
```
NSDocumentController.sharedDocumentController
```

Normally, you would never see or worry about the NSDocumentController instance; it'd just quietly do its work in the background. But we don't want it to be quiet—we want it to announce what it's doing (via puts) so that we can see how it fits in the app's flow of control. So, I've created a subclass, DocumentController (without the NS), for Fenestra to use.

Using such a subclass requires an odd little dance. The first time an NSDocumentController or subclass is created, it is installed as the instance that NSDocumentController.sharedDocumentController returns. If shared-

DocumentController is called before it has an object to return, it creates one. That new instance will be a plain NSDocumentController, not our subclass.

So, we have to create a DocumentController instance before any code would want to use it. An easy way to do that is to put the instance in the nib file:



It is at the moment that the nib file is fully loaded and DocumentController's awakeFromNib is called that we get the first line of our trace:

```
DocumentController awakes from nib.
...
```

Receiving newDocument

As discussed in Section 24.2, *The Responder Chain*, on page 316, the DocumentController receives the newDocument: message that Command-N generates. Because of the information in Info.plist, the newDocument method knows to create a Document object.

At this point, Fenestra has accomplished this:

```
DocumentController awakes from nib.
DocumentController asked to create a document.
...
```

The initialized Document

The Document is shown and edited in an NSTextView. To match that view's capabilities, our Document should contain not just a string of characters (an NSString), not just a string of characters that can be

changed (an NSMutableString) but a changeable string of characters where runs of characters can have *text attributes* like fonts associated with them (an NSMutableAttributedString). As the document initializes, it creates (at ❶ in the following code) an empty one of those for itself:

Download fenestra/document-based-spike/app/document/Document.rb

```
class Document < OSX::NSDocument
  def init
    $stderr.puts "#{self.class} springs into life!"
❶   @content = NSMutableAttributedString.alloc.init
❷   super_init
  end
end
```

Nothing has yet happened yet on the screen, but the log gets a new message:

```
...
DocumentController asked to create a document.
Document springs into life!
...
```

Creating the LogWindowController

At this point, we're at line ❷ in the preceding code snippet. That is, we're inside the Document's superclass's init method. Some object needs to take responsibility for the document's UI (still not yet created). It could be the Document itself. It could be Cocoa's NSWindowController. In this case, we want it to be an instance of our own NSWindowController subclass, which I'll name LogWindowController.

We put an LogWindowController in charge by overriding this method:

Download fenestra/document-based-spike/app/document/Document.rb

```
class Document < OSX::NSDocument
  def makeWindowControllers
    $stderr.puts "#{self.class} will now create its controller."
    main = LogWindowController.alloc.init
    addWindowController(main)
  end
end
```

As the name of the method implies, we could have several windows for several views into the document. Fenestra creates only one.

We've now proceeded this far:

```
...
Document springs into life!
Document will now create its controller.
...
```

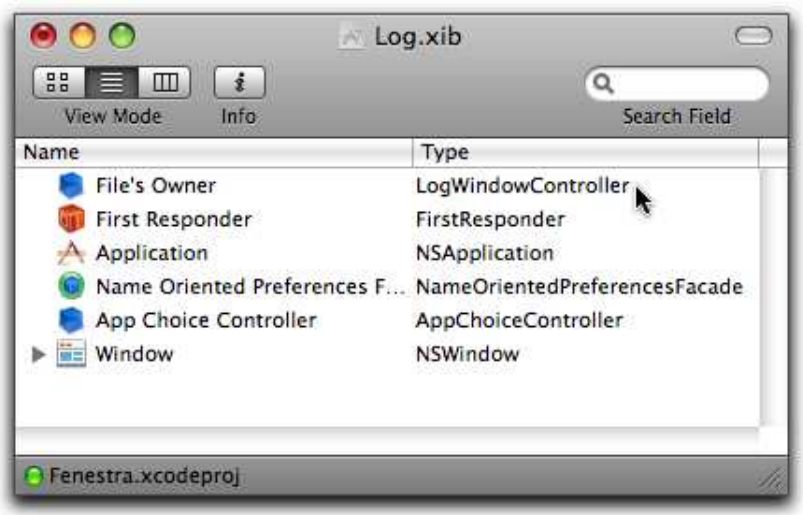
The Window Controller Starts to Create the Window

It's the newly created LogWindowController's job to create the window and all the views inside it. It starts that process like this:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def init
    $stderr.puts "#{self.class} springs into life!"
    $stderr.puts "#{self.class} will load the nib named 'Log'."
    initWithWindowNibName("Log")
    self
  end
end
```

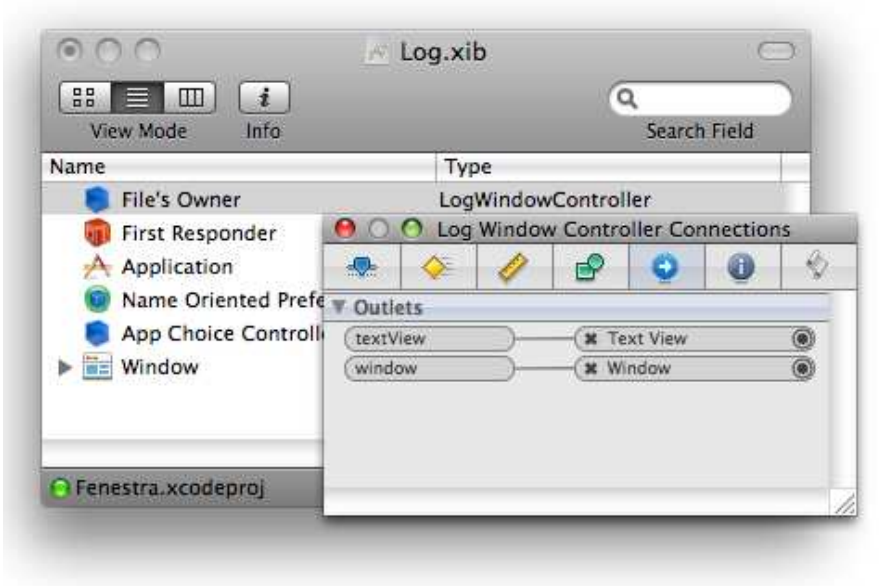
initWithWindowNibName doesn't actually load the nib file. The method just tells the controller what nib file to load when it needs it. Nevertheless, let's look inside the nib file now:



Notice that the LogWindowController isn't one of the names you see in the left column. That's because it has already been created. Since it will at some point load the nib file, though, it's represented inside the nib file by the pseudo-object File's Owner. (On the first line of the nib file, you can that see the File's Owner is defined to be a LogWindowController.) This representation lets outlets and action targets connect the LogWindowController and nib objects.

Because Fenestra objects use notifications for cross-controller communication, objects like the AppChoiceController don't need to use the File's

Owner as a target or have outlets to it. The File's Owner does, however, have two outlets to nib objects:



The textView outlet points to the text view that will contain translated notification text. It's declared here:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  ib_outlet :textView
  # ...
end
```

(In older versions of Fenestra, it was called log. While coding away on the spike, that name confused me once. In retribution, I changed it to something unambiguous.)

The window outlet is handled by the superclass. That outlet is actually a *lazy getter*. If you use it to retrieve the window and the nib file hasn't been loaded yet, it will be loaded at that point.

We're now at this point in the process:

```
...
Document will now create its controller.
LogWindowController springs into life!
LogWindowController will load the nib named 'Log'.
...
```

The Controller Learns About the Document

Now that the controller exists, the Document hands itself over to it by calling setDocument:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def setDocument(doc)
    $stderr.puts "#{self.class} has been given a pointer to #{doc.class}."
    super_setDocument(doc)
  end
end
```

After this, each object knows about the other. In particular, the LogWindowDocument can use its document accessor to ask the document for content to put in the textView. It mustn't do that yet, because @textView isn't connected to anything because the nib file has yet to be loaded.

We're now at this point in the scenario:

```
...
LogWindowController springs into life!
LogWindowController will load the nib named 'Log'.
LogWindowController has been given a pointer to Document.
...
```

Loading the nib file

Now that the Document and LogWindowController have been created, control returns to the DocumentController that started everything off. It now instructs the new Document to show all of its windows by calling its showWindows method. Here's what that method looks like:

Download fenestra/document-based-spike/app/document/Document.rb

```
class Document < OSX::NSDocument
  def showWindows
    $stderr.puts "#{self.class} has been asked to show its windows."
    super_showWindows
  end
end
```

As you see, the work is done by the superclass method. It sends the showWindow message to each controller it knows about. In our case, it knows about a single LogWindowController. (In Section 24.3, *Creating the LogWindowController*, on page 322, it used the method addWindowController to stash it away for future use.)

Here's LogWindowController's implementation of showWindow:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def showWindow(sender)
    $stderr.puts "#{self.class} has been asked to show its window."
    $stderr.puts "  (Is the Nib actually loaded yet? #{isWindowLoaded})"
    super_showWindow(sender)
  end
end
```

Again, the superclass method does the real work. You and I will likely never see the source for that method, but showing the window can be done only via the window getter, so the nib file starts loading.

At this point, we're here:

```
...
LogWindowController has been given a pointer to Document.
Document has been asked to show its windows.
LogWindowController has been asked to show its window.
  (Is the Nib actually loaded yet? false)

...
```

There's a subtlety here. In the picture of three Fenestra windows that started this chapter, you saw they were cascaded. That didn't work at first. A Google search showed me why. I had one little leftover window attribute:

Making the window visible when the nib file is loaded means that it appears before the NSWindowController has a chance to put in a good word for the virtues of cascading. So, I had to turn that attribute off to make cascading work.

The Controller and Window Are Connected

Even though a File Owner isn't defined in a nib file, it still receives the awakeFromNib message. Here's a piece of LogWindowController's implementation:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def awakeFromNib
    $stderr.puts "#{self.class} is told the Nib is loaded."
    $stderr.puts "#{self.class} now has an outlet to #{window.class}."
    # ...
  end
end
```

At this point, the controller has access to both the @textView and the document, so this would be a good time to move data from the latter to the former. First, though, here's a reminder of where we are:

```
...
LogWindowController has been asked to show its window.
  (Is the Nib actually loaded yet? false)
LogWindowController is told the Nib is loaded.
LogWindowController now has an outlet to OSX::NSWindow.
...
```

Initializing the View

Once everything is connected, it's the job of the LogWindowController to initialize the text view with the contents of the Document. These lines do that:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def awakeFromNib
    # ...
    @textView.textStorage.attributedString = document.content
    $stderr.puts "#{self.class} stuffs " +
                 "'#{document.content.to_s}' into text view."
    # ...
  end
end
```

I'll discuss them further in Section 24.5, *Editing*, on page 330.

We've now completed the saga:

```
...
LogWindowController is told the Nib is loaded.
LogWindowController now has an outlet to OSX::NSWindow.
LogWindowController stuffs '' into text view.
...
```

Some Differences Between the Old and New Fenestra

In the old version of Fenestra, there was a class called of MainWindowController. You can see it in Figure 8.2, on page 102. *None* of the code there is relevant to our new Fenestra:

- It put "No App" in the title, but a new document should instead say "Untitled."
- When Fenestra was fenestrating another app, it put the name of the other app in the title, but a document window should give the name of the document.
- It terminated the app when the window was closed, but that's incorrect in a multiwindow application.

The LogWindowController has all the correct behavior, just by virtue of inheriting from NSWindowController.

Since the LogWindowController would therefore be empty of nondebug code, I decided to put all the code that worked with the text view into it. That code used to be in a separate controller, LogController, but now that the text view is really a view into a document, it makes sense for the NSWindowController that's created by the document to control it.

You'll see in Section 24.5, *Editing*, on page 330, how the LogController from older versions was changed in this new Fenestra.

24.4 Opening and Saving Documents

Now things start to get simple. Once we tell NSDocument how to turn its in-memory version to a disk-suitable version, we get all the behavior we expect from menu items like Open, Close, Save, and Save As. Double-clicking a file that ends in .fenestra will start Fenestra. You can drag such a file from the Finder onto Fenestra's Dock icon, and a document window will spring into existence. And so on.

An easy way to create disk-ready data is to use the same archive/unarchive mechanism we've been using since Section 11.2, *Archiving*, on page 134. However, I want to show you something new. NSAttributedStrings have methods that support easy conversion from and to RTF. If we save the content of the Document as RTF, then other applications can edit it.

These two methods support conversion:

Download fenestra/document-based-spike/app/document/Document.rb

```
class Document < OSX::NSDocument
  def dataOfType_error(type, error)
    content.objc_send(:RTFFromRange, NSRange.new(0, content.length),
                      :documentAttributes, nil)
  end

  def readFromData_ofType_error(data, type, error)
    string = NSMutableAttributedString.alloc
    @content = string.objc_send(:initWithRTF, data,
                                :documentAttributes, nil)
    true
  end
end
```

Both of the methods take error arguments (which would be ObjcPtrs). Since this is a spike, I haven't bothered to worry about how the methods could fail.

The methods also take a type argument. Some apps (TextEdit, for example) can operate on different kinds of files with different extensions. Fenestra operates only on *"fenestra"* files, so I ignore the argument.

Try This Yourself

I've implemented two menu items under Fenestra's Format menu: Show Colors and Underline.[2] Put some colored, underlined text in the text view, save the file, and open it with TextEdit. Does it work?

I have a wild imagination, but I still can't imagine firing up Fenestra and reopening a previously saved log. I *can* imagine opening a saved log in TextEdit or some even more capable editor, annotating it, and mailing it off as a bug report. So, I'd rather Fenestra files were plain-old RTF files that open in TextEdit.

2. Show Colors invokes the action orderFrontColorPanel on the File's Owner target (NSApplication). Underline invokes underline on the First Responder (NSTextView).

That can be done with one change to Info.plist. Change *CFBundleTypeExtensions* to contain *rtf* instead of *fenestra*. Run Fenestra, save a file, double-click it in the Finder, and see what happens.

24.5 Editing

You might recall that an NSTextView has an accessor named textStorage. (It was introduced on page 52.) An NSTextStorage is an NSMutableAttributedString that contains a series of NSLayoutManagers. An NSLayoutManager controls the display of a portion of the NSTextStorage. For our purposes, we don't need to care about layout managers; we accept whatever the default layout is.

Back in Section 24.3, *The initialized Document*, on page 321, the document was initialized with an empty NSAttributedString. The LogWindowController put that string into the text view after the nib file was loaded. That code (again) is here:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def awakeFromNib
    # ...
❶   @textView.textStorage.attributedString = document.content
    $stderr.puts "#{self.class} stuffs " +
                 "'#{document.content.to_s}' into text view."
    # ...
  end
end
```

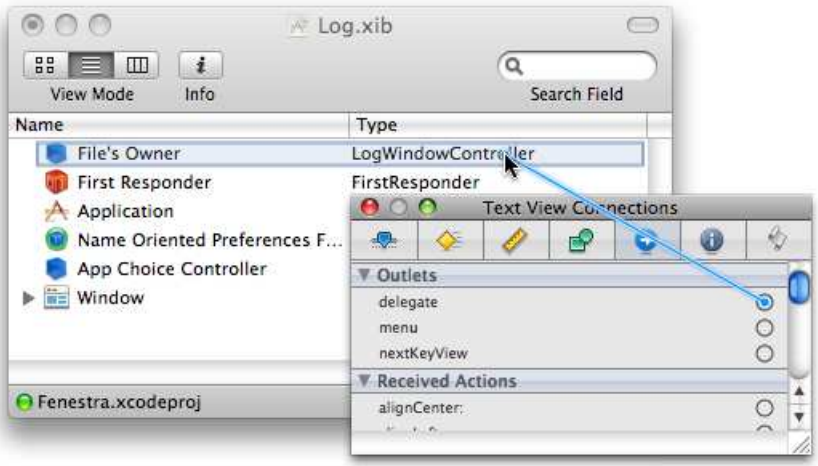
I find that code misleading. It *looks* as if an NSTextStorage has an @attributedString property that's getting set. Not so. I meant it when I wrote earlier that an NSTextStorage *is* an NSMutableAttributedString. This code replaces all the characters and attributes of @textView.textStorage with those of document.content. I mention this now to head off some unclarity later.

There are two scenarios in which the document changes. I'll treat them in turn.

User Edits

In this scenario, the user makes a change: she types, pastes, changes text color, underlines with the Underline menu item, or whatever.

A text view informs its delegate whenever it's changed by the user, so the first step in making editing work is to make the LogWindowController the text view's delegate:



When the text view makes a change, the controller's textDidChange method is called. It's now the controller's job to pass the change along to the document. Here's the code:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
class LogWindowController < OSX::NSWindowController
  def textDidChange(unused_notification)
    document.content = @textView.textStorage
  end
end
```

On first blush, that confused me. We're replacing what used to be an NSMutableAttributedString with an NSTextStorage. That confusion prompted me to discover that an NSTextStorage is an NSMutableAttributedString subclass.[3]

3. You might think a slick way to avoid needing this code is to have the text view and the document share the same NSTextStorage. That way, changes made by the text view would just appear in the document. That won't work. The text view keeps getting new NSTextStorage objects. You can turn on logging that shows the changing object identities—just remove comments from some lines in Document's content= method.

The document's content= method isn't just a simple setter, though:

Download fenestra/document-based-spike/app/document/Document.rb

```
class Document < OSX::NSDocument
  def content=(new_content)
    updateChangeCount(NSChangeDone)
    @content = new_content
  end
end
```

The updateChangeCount informs the framework that the document has changed. If you try to close it, the framework will provide the usual "Do you want to save the changes you made in the document?" prompt.

There's no need for your code to do anything when the document is saved. The "dirty bit" is automatically turned off.

Programmatic Edits

The text view can be changed programmatically in response to notifications sent by a Translator (as well as by other objects). In earlier versions of Fenestra, text from notifications was put into the text view with code like this:

Download fenestra/fit-and-finish/app/main-window/LogController.rb

```
on_local_notification AppFactAvailable do | notification |
  @log.addLine(notification[:message])
end
```

Recall that in Section 4.5, *Reopening Objective-C Classes*, on page 73, I tweaked NSTextView to define addLine like this:

Download fenestra/document-based-spike/app/util/NSPatches.rb

```
class NSTextView
  def addLine(string)
    string += "\n"
    at_end = NSRange.new(textStorage.length, 0)
    replaceCharactersInRange_withString(at_end, string)
  end
end
```

It turns out that methods like replaceCharactersInRange_withString do not inform the delegate of the change, so I rewrote the on_notification methods like this:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
on_local_notification AppFactAvailable do | notification |
  addLine(notification[:message])
end
```

. . . and defined addLine to note the change:

Download fenestra/document-based-spike/app/log-window/LogWindowController.rb

```
def addLine(line)
  @textView.addLine(line)
  textDidChange(:irrelevant)
end
```

And that's all there is to it.

24.6 Learning More

Document-based applications can be much richer than what I have shown you in these chapters. To get started learning more, start with Apple's *Document-Based Applications Overview* [App08j]. You can find the source for TextEdit in /Developer/Examples/AppKit/TextEdit. Many of the other developer examples are document-based as well. The RubyCocoa examples in /Developer/Examples/Ruby/RubyCocoa tend have minimal use of document-based features.

Cocoa text support is almost outlandishly rich. Need to have text flowing in two different directions within a line? It can be done. The two documents to start with are Apple's *Text System Overview* [App08y] and *Text System User Interface Layer Programming Guide for Cocoa* [App08z].

Chapter 25

MacRuby

MacRuby will eventually replace RubyCocoa. At the time I write (July 2009), I doubt it's ready to do that, but I certainly do encourage you to try it.

Take a look at Figure 25.1, on the following page. It shows, in cartoon form, the difference between RubyCocoa and MacRuby. In RubyCocoa, what I've been calling "the Ruby universe" and the "Objective-C universe" are on the left and right, respectively. Each universe is made up of two parts. The first is a *virtual machine* (VM). I like to think of a VM as a DSL tailored to the implementation of a particular language. Recall how in Section 8.1, *A DSL for Notifications*, on page 101, we wrote a more Rubylike version of notifications. With it, we can write code like this:

```
on_local_notification AppChosen do | notification |
  # ...
end
```

. . . and the DSL takes care of most of the work of adding observers. Similarly, a Ruby VM provides a language tailored for implementing Ruby. It translates from that friendly-to-a-Ruby-implementor's language into all the detailed instructions that an Intel chip requires.[1]

The second part of a universe consists of all the thousands of objects your Ruby or Objective-C program creates.

1. The phrase "virtual machine" comes from a different slant on the metaphor. Instead of thinking of the virtual machine as a language translator, we think of it as a software implementation of a chip made just for running Ruby. In the metaphor, Ruby thinks it's running on some bizarre-yet-pleasing raw hardware.

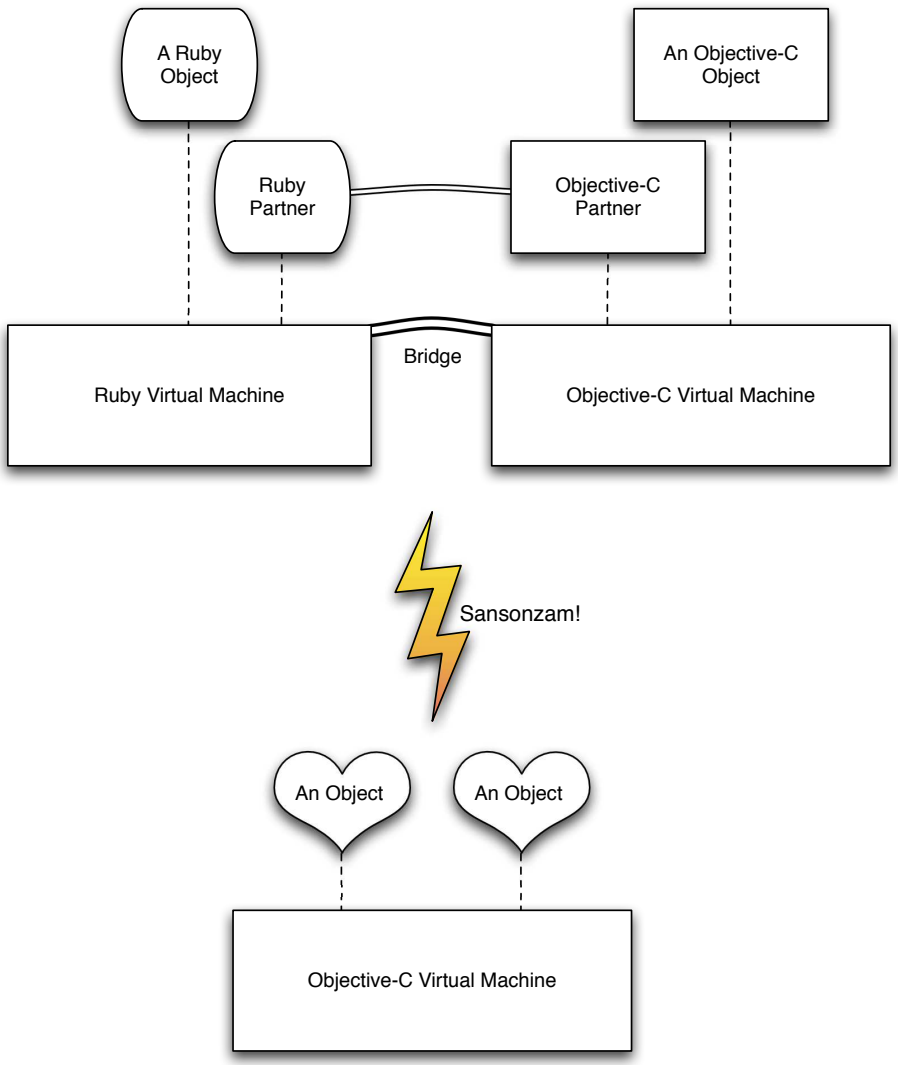


Figure 25.1: RubyCocoa and MacRuby

RubyCocoa connects (or "bridges") the two universes in three ways.

- When a Ruby object that descends from NSObject is created, it is partnered with a newly created Objective-C object. The latter object handles messages sent to it from inside the Objective-C universe. If it can't handle a message, it passes it on to its Ruby partner. The Ruby partner does the same sort of thing with messages coming from the Ruby universe.
- When a pure-Ruby object is passed into Objective-C code, it's converted into some sort of Objective-C object. For example, in the following, *"hi"*—a ruby String—is converted into an NSString before it's given to hasPrefix:

```
irb(main):008:0> some_nsstring.hasPrefix("hi")
=> #<NSCFString "hi">
```

- Objective-C objects being passed into the Ruby universe are partnered with a Ruby object that will forward messages to its partner.

MacRuby does away with partnered objects. All objects, those written in Objective-C and those written in Ruby, use the Objective-C VM. (That means that the Ruby implementation has been rewritten to use the Objective-C DSL.) This has two big advantages:

- It avoids the app slowdown caused by RubyCocoa's object creation and the forwarding of messages from universe to universe.
- When writing code, you don't have to wonder whether the string you're working with might be a visitor from another universe. Whether the string was created in the Ruby universe as a String or the Objective-C universe as an NSMutableString, it will respond to all the same messages (whatever messages are accepted by either String or NSMutableString).

MacRuby is one of several Ruby implementations that run on different virtual machines. JRuby runs on the Java virtual machine, and IronRuby runs on .NET. Just as MacRuby makes it easy to use Objective-C objects, JRuby makes it easy to use Java objects, and IronRuby makes it easy to use C# objects. I expect all of the implementations to remain compatible with the "official" Ruby in the sense that they'll be able to run any program it runs.

25.1 Getting MacRuby

You download MacRuby from http://www.macruby.org. The download package is helpful. It adds MacRuby projects to Xcode's New Project dialog box, and it puts examples in /Developer/Examples/MacRuby. In the following examples, I'm using MacRuby 0.3.

25.2 MacRuby Basics

All of the MacRuby command-line apps are their plain-Ruby equivalents with a "mac" prepended. Here, for example, is macirb:

```
$ macirb
>>
```

I'll use macirb to demonstrate MacRuby features. Keep in mind that some of these might change before the first production release.

Classes, Objects, and Nonobjects

MacRuby unifies common Ruby classes with their Objective-C equivalents. For example, String and NSMutableString are *identical*:

```
>> NSMutableString.object_id == String.object_id
=> true
```

Here are some of the implications:

```
>> "foo".class
=> NSMutableString        # NSMutableString is the default class name.

>> String.new.class
=> NSMutableString        # ... even if the object is created from String

>> "foo".is_a? String     # Old code using is_a? will still work.
=> true
>> "foo".is_a? NSMutableString
=> true
```

There are issues that class unification doesn't resolve. You still need to be able to create objects, like NSRange, that wrap Objective-C structs. And it remains a fact that Objective-C thinks the machine integer 0 means "false."

In its raw form, MacRuby doesn't help with those issues:

```
>> NSRange.new(0, 12)
NameError: uninitialized constant NSRange
        from (irb):4
        from /usr/local/bin/macirb:12:in `<main>'
>> "string".hasPrefix("hurm")
=> 0
```

To get the behavior you're used to, you need to load the Cocoa framework:

```
>> framework 'Cocoa'
=> true
```

Once you do that, you get the same behavior this book has taught you to expect from RubyCocoa:

```
>> NSRange.new(0, 12)
=> #<NSRange location=0 length=12>
>> "string".hasPrefix("hurm")
=> false
```

You want the Cocoa framework anyway, since it has the tasty classes that let you make spiffy UIs:

```
>> NSMutableAttributedString.alloc.init
=> #<NSConcreteMutableAttributedString:0x17fc5a0>
>> NSButton.alloc.init
=> #<NSButton:0x14713c0>
```

You can use new as an abbreviation for alloc.init:

```
>> NSMutableAttributedString.new
=> #<NSConcreteMutableAttributedString:0x17e6030>
```

Calling and Defining Methods

MacRuby has added to Ruby syntax to make calling Objective-C methods more convenient, as shown at ❶.

```
>> s = "hello, sailor"
=> "hello, sailor"
>> r = NSRange.new(s.index('sailor'), 6)
=> #<NSRange location=7 length=6>
❶ >> s.replaceCharactersInRange r, withString: "world"
=> nil
>> s
=> "hello, world"
```

Here's another example of the method-calling syntax, this time with parentheses:

```
>> r = NSRange.new(0, 6)
=> #<NSRange location=0 length=6>
❶ >> s.replaceCharactersInRange(r, withString: "goodbye cruel")
=> nil
>> s
=> "goodbye cruel world"
```

My favorite thing about this syntax is that you define (❶) methods the same way you use (❷) them:

```
>> class Definitions
❶ >>   def printThis first, andThis: second
>>     puts first, second
>>   end
>> end
=> nil
❷ >> Definitions.new.printThis 1, andThis: 2
1
2
=> nil
```

Notice that Definitions didn't mention NSObject on the **class** line. That means it descends from Object, which is identical to NSObject:

```
>> Object.object_id
=> -1606472512
>> NSObject.object_id
=> -1606472512
>> Definitions.ancestors
=> [Definitions, NSObject, Kernel]
```

Unlike RubyCocoa, overriding methods call their superclass in the normal Ruby way. Here's an example:

Download macruby/subclass.rb

```
class SubClass < Definitions
  def printThis first, andThis: second
    sorted = [first, second].sort
❶   super sorted[0], sorted[1]
  end
end
```

Notice that the keyword **andThis:** isn't used in the super call.

Here, macirb shows that the subclass method executes and then calls the superclass:

```
>> SubClass.new.printThis 3000, andThis: 0
0
3000
=> nil
```

25.3 A MacRuby Checklist

Here's a checklist for converting a RubyCocoa program to a MacRuby program. I do not claim it's complete.

1. Both calls that use objc_send and the infix-underscore style (convertPoint_fromView) need to be changed to the new syntax.

2. Instances of to_ns and to_ruby need to be removed.

3. In some places there may be tests for Objective-C booleans (0 and 1) instead of true and false. That code will now be surprised when it no longer gets integers.

4. The module OSX no longer exists. That means code like this:

   Download fenestra/fit-and-finish/app/util/NSPatches.rb

   ```
   module OSX
     class NSTextView
       def clear
         # ...
       end
     end
   end
   ```

   . . . no longer adds on to an existing class. It now creates a new one in a new module.

5. ib_outlet has been deprecated. You're supposed to use attr_writer or attr_accessor instead.[2]

6. ib_action has been deprecated. In MacRuby, an action is any method whose single argument is named sender.

7. Notification selectors now must include their trailing colons. Whereas you could before write this:

   ```
   center.objc_send(:addObserver, self,
                    :selector, :handle_notification,  # symbol or string
                    :name, "name",
                    :object, nil)
   ```

   . . . you must now write this:

   ```
   center.addObserver self,
          selector: 'handle_notification:'      # string ending in colon
          name: "name",
          object: nil
   ```

8. Files that requireosx/cocoa should be changed to this:

   Download fenestra/macruby/rb_main.rb

   ```
   framework 'Cocoa'
   ```

2. I miss ib_outlet. I like a convention that calls out which attributes are used by Interface Builder.

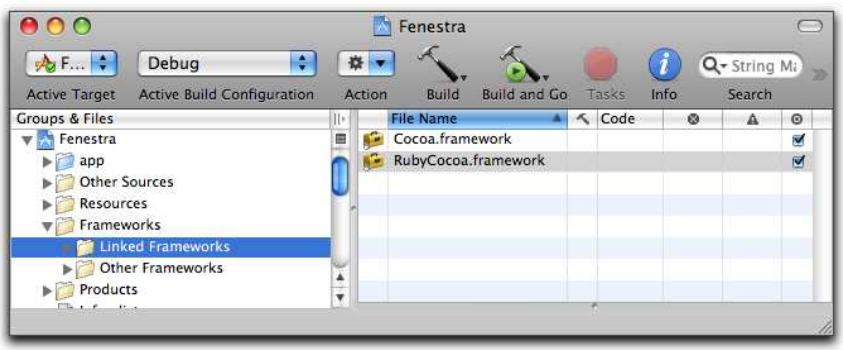
9. The Objective-C code that starts an app should be changed to look like this:

   Download fenestra/macruby/main.m

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

   int main(int argc, char *argv[])
   {
       return macruby_main("rb_main.rb", argc, argv);
   }
   ```

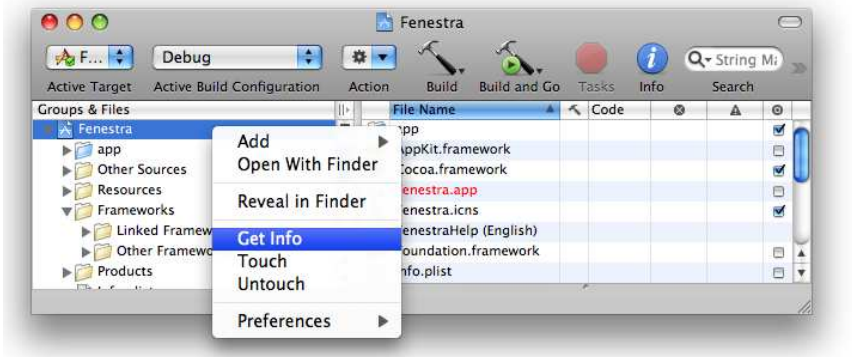
   Xcode creates this code when you create a MacRuby project.

10. The RubyCocoa framework should be deleted and the MacRuby framework added. The RubyCocoa framework is listed under the Linked Frameworks group in Xcode's project browser, as shown here:

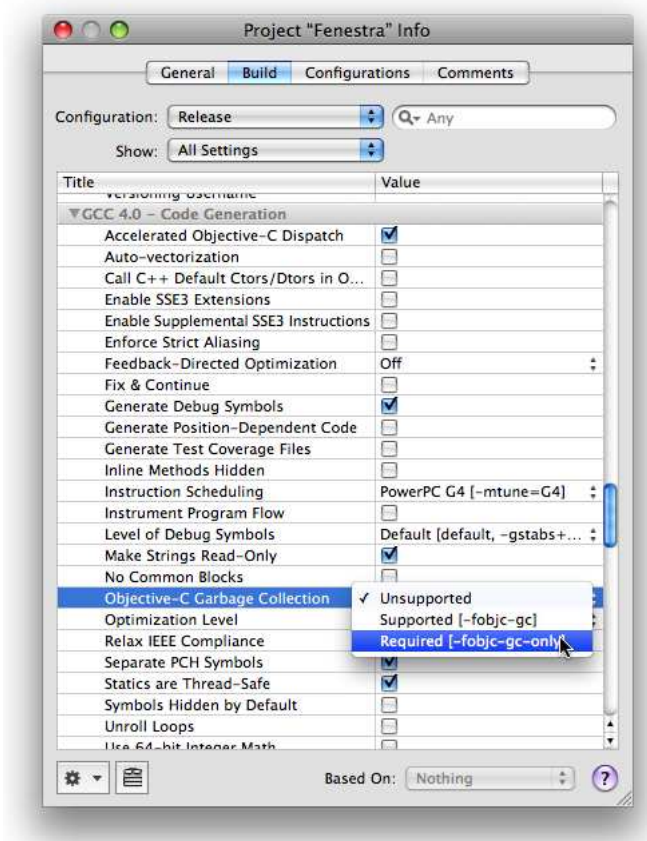


   Add the framework by right-clicking the Linked Frameworks icon and choosing Add > Existing Frameworks. When you add the framework, the Open File dialog box is likely to show you /System/Library/Frameworks. MacRuby is actually under /Library/Frameworks.

11. The Objective-C part of your app must be told to use garbage collection. To do that, right-click the name of the project in the project browser, and choose Get Info.



A window will pop up. Pick the Build tab, and then scroll down to GCC 4.0 – Code Generation. Change Objective-C Garbage Collection to Required, as shown here:



You have to make that change separately for both the Debug and Release configurations (which you can choose at the top of the window).

25.4 What Now?

Since I'll continue to work on the MacRuby version of Fenestra, I don't want to describe the other changes I've made to it here. Instead, I'll periodically update the source in fenestra/macruby. Changes will be marked with a comment beginning #PORT:. The comment will explain and justify the change.

I encourage you to check up on Fenestra as a way of deciding when you'll be ready to switch your development work from RubyCocoa to MacRuby. You'll also find the MacRuby-Devel list[3] to be a good gauge of MacRuby status. Until then: bring that RubyCocoa love!

3. http://lists.macosforge.org/mailman/listinfo.cgi/macruby-devel

Part VII

Reference

Chapter 26

The Objective-C Bridge and Bridge Metadata

You saw throughout the book how data was converted as it passed across the bridge between the Objective-C and Ruby universes. The conversions happened automatically, without our help. Sometimes, though, you'll have to do some explicit setup to help RubyCocoa make the conversions. That's done with *bridge metadata*, the topic of this chapter.

26.1 An Unexpected Return Value

Here's a new Objective-C object with a method that checks the length of a string:

Download rubycocoa-oddities/test/Bridged.m

```
@implementation Bridged

❶ - (BOOL)hasEvenLength: (NSString *)string
{
    int length = [string length];
    int remainder = length % 2;
    if (remainder == 0)
      return YES;
    else
      return NO;
}

@end
```

Here it is in use:

```
irb(main):007:0> require 'Bridged'
=> true
irb(main):008:0> Bridged.alloc.init.hasEvenLength("ab")
=> 0
irb(main):009:0> Bridged.alloc.init.hasEvenLength("ab")
=> 1
```

Notice that the return value, declared as (BOOL) at ❶, returns the integers 0 and 1. That behavior isn't like what we've seen in the past. Here, for example, is an NSFileManager method, also declared to return (BOOL):

```
- (BOOL)fileExistsAtPath:(NSString *)path
```

Yet its return value is converted from an integer to a Ruby boolean:

```
irb(main):002:0> manager = NSFileManager.defaultManager
=> #<OSX::NSFileManager:0x290324 class='NSFileManager' id=0x2adc30>
irb(main):003:0> manager.fileExistsAtPath("/System/Library/Frameworks/")
=> true
```

Since it's possible for RubyCocoa to convert integers to Ruby booleans for some methods, why doesn't it for hasEvenLength? It's because RubyCocoa has only runtime information about it.

26.2 What Information Can Be Found at Runtime?

Like Ruby, Objective-C can discover certain information about methods at runtime. Let's see what we can find out about fileExistsAtPath. Here's how to fetch information about a method:

```
irb(main):003:0> m = NSFileManager.instanceMethodSignatureForSelector(↩
'fileExistsAtPath:')
=> #<OSX::NSMethodSignature:0x29338a class='NSMethodSignature' id=0x1b43fb0>
```

An NSMethodSignature is roughly (*very* roughly) equivalent to the result of Ruby's Method#arity). So, for example, you can ask for the number of arguments:

```
irb(main):007:0> m.numberOfArguments
=> 3
```

Three? Why three?

All Objective-C methods have two "hidden" arguments. The first is self, and the second is the selector for the method (roughly, its name).[1] So,

1. I don't know of any way to access the second variable from Ruby code.

what we see as the only argument to the method is actually the third. And what do we know about it?

```
irb(main):009:0> m.getArgumentTypeAtIndex(2)
=> "@"
```

That character is a special code that means "object." To Ruby programmers, that seems totally redundant: all Ruby arguments are objects because everything is an object. But that's not true of Objective-C, which has types like "machine integer" as well as "object."

And what's the return type?

```
irb(main):006:0> m.methodReturnType
=> "c"
```

c tells us the return value is a small machine integer (specifically, a value from -128 to 127). You know and I know that fileExistsAtPath returns 0 to mean "the file doesn't exist" and some nonzero number to mean "it does so exist," but Objective-C hasn't read the Cocoa API documentation and doesn't have the common sense to deduce that the method name is a question. It leaves matters of interpretation entirely up to the caller. We need a way to explain our desired interpretation to RubyCocoa.

26.3 Supplementing Runtime Information

We expect Ruby to maintain a distinction between truth values and numbers. Unfortunately, as we've seen, it can't rely on the Objective-C runtime. Instead, the good folk at Apple have provided *bridge metadata* files that Ruby (and other languages) can use to get more detail than is available from the runtime. These files have the suffix bridgesupport. The bridge support file for NSFileManager (and many other Cocoa classes) is /System/Library/Frameworks/Foundation.framework/Resources/BridgeSupport/FoundationFull.bridgesupport. Here's the snippet that describes fileExistsAtPath:

```
<class name='NSFileManager'>
    ...
    <method selector='fileExistsAtPath:'>
        <arg name='path' declared_type='NSString*' type='@' index='0'/>
        <retval declared_type='BOOL' type='B'/>
    </method>
    ...
</class>
```

The **retval** clause declares the return value to be a small integer that's interpreted as a truth value. Because Foundation is a framework, RubyCocoa loads up the metadata when it loads the framework.

Some frameworks are loaded automatically with RubyCocoa. You can load others with require_framework. For example, here's how to load the framework that accesses iCal:

```
741 $ irb
irb(main):001:0> c = CalCalendarStore.defaultCalendarStore
NameError: uninitialized constant CalCalendarStore
        from /System/Library/Frameworks/RubyCocoa.framework...
        from (irb):1
irb(main):002:0> # Oops...
❶ irb(main):003:0* require_framework 'CalendarStore'
=> true
irb(main):004:0> c = CalCalendarStore.defaultCalendarStore
=> #<OSX::CalCalendarStore:0x293722 class='CalCalendarStore' id=0x1ba05b0>
irb(main):005:0> puts c.calendars.collect { | c | c.title }.to_ruby
Brian Work
Dawn work
Home
Clinics
Dawn travel
=> nil
```

26.4 Our Own Private Metadata

I have hand-coded a bridge support file, Bridged.bridgesupport, for the Bridged class. It contains this:

Download rubycocoa-oddities/test/Bridged.bridgesupport

```
<class name='Bridged'>
  <method selector='hasEvenLength:'>
      <arg name='string' declared_type='NSString*' type='@' index='0'/>
      <retval declared_type='BOOL' type='B'/>
  </method>
</class>
```

The single argument is declared as an object ('@'). The return value is *'B'* as it was for fileExistsAtPath.

Here's the test to see whether RubyCocoa really truly converts the small integer values to Ruby booleans:

Download rubycocoa-oddities/test/bridged-tests.rb

```
require 'Bridged'
OSX.load_bridge_support_file 'Bridged.bridgesupport'

class BridgedTests < Test::Unit::TestCase
  should "produce booleans, not integers" do
    @sut = Bridged.alloc.init
    assert { @sut.hasEvenLength('1') == false }
    assert { @sut.hasEvenLength('12') == true }
  end
end
```

If you run that test, you should see it pass.

26.5 Finding Out More

You can find a basic description of the bridge in *Ruby and Python Programming Topics for Mac OS X* [App08w], particularly the section titled "Generating Framework Metadata." The full description of the bridge support file's XML format can be had by typing this to a shell prompt:

```
$ man BridgeSupport
```

(man is a venerable old Unix documentation format. In the jargon, the previous command provides you with the "BridgeSupport man page.")

The cryptic type symbols ("^@" and the like) are documented in the "Type Encodings" section of *Objective-C 2.0 Runtime Programming Guide* [App08s].

Most of the content of a bridge support file can be generated automatically using the gen_bridge_metadata command. It has a man page. It's usually used to add bridge metadata to entire frameworks.

Chapter 27

The Underpinnings of Cocoa Bindings: Key-Value Coding and Observing

Earlier discussions of Cocoa bindings were incomplete. Chapter 14, *Setting Up Bindings with Code*, on page 185 showed you how to bind, say, an NSTextField's value to a rooted keypath.[1] But I haven't shown you how to implement either bound objects like NSTextField or the properties they bind to. This chapter fills that gap by explaining how all the pieces of the binding mechanism work together.

27.1 Requirements

In Cocoa bindings, a bound object must do three things:

- Implement bind_toObject_withKeyPath_options and, in it, record the rooted keypath.
- Respond to a message sent it whenever an observed property key changes its value.
- Change the value of the observed key at the end of the rooted keypath when appropriate. Most bound objects are views, so "when appropriate" typically means when a person has done something through the user interface.

1. For a refresher on terminology, see Section 14.1, *Oh No! Terminology!*, on page 185.

An object possessing an observed property key needs to do only one thing:

- Declare or implement the property's setter in a way that informs bound objects of changes.

This chapter will show two objects that do all these things.

27.2 Our Goal

We want to write a script that executes these steps:

1. It creates a root object with a property (that is, an observed property key "property"). Afterward, the script prints this:

   ```
   Step 1: Creating an object to be root.
           root.property is "initial".
   ```

2. It creates a bound object with a bound name of "some bound name." I've chosen a name that's not a valid Ruby identifier to emphasize that bound names are not the same as properties.

   After creating the bound object, the script prints this:

   ```
   Step 2: Creating a bound object to track the root.
           bound_object holds nil for 'some binding name'.
   ```

3. The two objects are bound together. Immediately, the root's property value migrates into the bound object:

   ```
   Step 3: Bind the objects.
           bound_object holds #<NSCFString "initial"> for 'some bin↩
   ding name'.
   ```

   Notice that the Ruby string has been converted to an NSString. That's because it passes through Objective-C code on its way to the bound object.

4. Next, the root's property is updated, and the change propagates to the bound object:

   ```
   Step 4: Updating root property to "new".
           bound_object holds #<NSCFString "new"> for 'some binding↩
   name'.
   ```

5. Finally, we call the bound object's upcase method. (Pretend it was called because it's the action method for a button in the UI and someone just clicked that button.) upcase converts the data associated with "some binding name" to capital letters. Afterward, the bound object changes property's value.

Download key-value/howtos/kv-binding-by-hand.rb

```
class Root < NSObject

  attr_reader :property

  def property=(value)
    willChangeValueForKey(:property)
    @property = value
    didChangeValueForKey(:property)
    @property
  end

  def initWithValue(value)
    @property = value
    self
  end
end
```

key-value/howtos/kv-binding-by-hand.rb

Figure 27.1: A root object

```
    Step 5: Pretending to click upcase button.
            bound_object holds #<NSCFString "NEW"> for 'some binding ↩
    name'.
            root.property is #<NSCFString "NEW">.
```

27.3 Declaring Observed Properties

I'll begin with the root object, since it's simpler. You can see a cumbersome way of defining an observed property[2] in Figure 27.1. The willChangeValueForKey method (❶) records the old value (which the bound object can ask for), and didChangeValueForKey (❷) propagates the new one.

Writing code like this is boring, so there's a shorthand way of declaring an observed property:

Download key-value/howtos/kv-binding-by-hand.rb

```
kvc_accessor :property
```

2. Apple calls a properly defined observed property *KVO-compliant*, where KVO stands for "key-value observing." You can learn more about this in Section 27.4, *Observing Changes*, on the following page.

The kvc prefix stands for "key-value coding," which you'll learn about in Section 27.6, *Changing the Value of an Observed Key*, on page 356.

27.4 Observing Changes

When didChangeValueForKey runs, it will call a particular method in the bound object. This may remind you of the way notifications are delivered to objects, but notification observers get to pick which method gets called. In bound objects, it's always the same one:

Download key-value/howtos/kv-binding-by-hand.rb

```
class BoundObject < NSObject
  # ...
  def observeValueForKeyPath_ofObject_change_context(keypath, root,
                                                     change, context)
    # ...
  end
```

Before describing the code within that method, let me talk about BoundObject's two instance variables:

Download key-value/howtos/kv-binding-by-hand.rb

```
class BoundObject < NSObject

  def init
    @data = {}
    @tracker = BindingTracker.new
    self
  end
```

This script binds only one name. It would be easy enough to associate that name with an instance variable (maybe @some_binding_name), but I want to push back against the natural assumption that bindings are just a variation on instance variable getters and setters. So, I'll use a hash, @data, to store a name-to-value correspondence.

As you'll see later, the bound object needs to translate from a rooted keypath to its bound name, or vice versa, so I store those correspondences in @tracker, a BindingTracker. That class's implementation is painfully simple, so I'll show only one directly relevant bit (later).

With that out of the way, here's a first bit of observation code:

Download key-value/howtos/kv-binding-by-hand.rb

```
def observeValueForKeyPath_ofObject_change_context(keypath, root,
                                                   change, context)

  puts "Received update from #{root}://#{keypath}:"
  puts change.to_ruby.inspect
end
```

Here is what's printed when the binding is made and the bound object receives the initial value:[3]

```
Received update from <Root: 0x571020>://property:
{"new"=>"initial", "kind"=>1}
```

In this application, the "kind" will always be 1 (symbolically NSKeyValueChangeSetting), meaning "a value has changed." If the observed property were an array or some other collection, you could also be informed of insertions, deletions, or element replacement.

After a change to the property, the bound object will receive something like this:

```
Received update from <Root: 0x571020>://property:
{"new"=>"new", "kind"=>1, "old"=>"initial"}
```

In this case, the change description also contains the value stashed when the root object's setter called willChangeValueForKey.

Here's what the method does with the information:

Download key-value/howtos/kv-binding-by-hand.rb

```
def observeValueForKeyPath_ofObject_change_context(keypath, root,
                                                   change, context)
  # ...
  bound_name = @tracker.bound_name_for(root, keypath)
  @data[bound_name] = change['new']
end
```

It finds the bound name (stored by bind_toObject_withKeyPath_options, as you'll see next) and uses that to save the new value of the observed property key.

3. You may notice I didn't print the context argument. You can see the details of its use in key-value/howtos/key-value-observing-context-arg.rb.

27.5 Implementing bind_toObject_withKeyPath_options

The bound object needs to implement bind_toObject_withKeyPath_options to associate the bound object with the rooted keypath it observes. That's rather straightforward:

Download key-value/howtos/kv-binding-by-hand.rb

```
def bind_toObject_withKeyPath_options(bound_name, root, keypath, options)

❶   @tracker.remember(bound_name, root, keypath)

❷   root.objc_send(:addObserver, self,
                   :forKeyPath, keypath,
                   :options, NSKeyValueObservingOptionInitial |
                             NSKeyValueObservingOptionNew |
                             NSKeyValueObservingOptionOld,
                   :context, nil)
end
```

At ❶, it remembers that a change in this particular rooted keypath affects the bound name. At ❷, it asks the root to inform it (the bound object) when anything in the keypath changes. It chooses the following options:

- The bound object wants to be informed immediately of the observed property key's value. (By default, it's informed only after a change.)
- It wants to know the new value after a change.
- It wants to know the old value (even though, in this case, it's not used).

There's another option we didn't use. NSKeyValueObservingOptionPrior causes the bound object to be informed twice for each change: once at the moment of willChangeValueForKey (containing the old value) and then again at the moment of didChangeValueForKey.

Notice all these options are bitwise-or'd together.

The context is an arbitrary value that, as we've seen, is passed to the observer when there's a change in the rooted keypath. Before using it in one of your apps, see key-value/howtos/key-value-observing-context-arg.rb—using it is not as simple as putting an object in one side and having it pop out the other.

27.6 Changing the Value of an Observed Key

All that remains is reacting to the fake button press. That's done with this code in the bound object:

Download key-value/howtos/kv-binding-by-hand.rb

```
ib_action :upcase do | sender |
  older = @data['some binding name']
  newer = older.upcase
  @data['some binding name'] = newer
❶ @tracker.propagate('some binding name', newer)
end
```

It first updates @data. That's no different than updating in response to a change delivered to observeValueForKeyPath_ofObject_change_context. But then, at ❶, it propagates the change to the object at the end of the rooted keypath. In this case, that object happens to be the root itself, but the bound object doesn't know that. Here's the implementation of the BindingTracker's propagate method:

Download key-value/howtos/kv-binding-by-hand.rb

```
class BindingTracker
  # ...
  def propagate(bound_name, value)
    target = rooted_keypath_for(bound_name)
    target.root.setValue_forKeyPath(value, target.keypath)
  end
```

It first looks up the two components of the rooted keypath (previously stashed away by bind_toObject_withKeyPath_options). It then uses a new method, setValue_forKeyPath, to update the property. That method sets the value of the property key at the end of the keypath. For it to work, two things must be true:

- All the components of the keypath, except the last, must implement valueForKey. That's easy for Ruby subclasses of NSObject, since NSObject behaves like it contains this method:

  ```
  def valueForKey(key)
    self.send(key)
  end
  ```

  So, all that's needed is a method named key. It could be created by attr_reader or kvc_accessor, or it could be written explicitly with **def**.

- The last component of the keypath has to implement setValue_forKey. That's also easy, since NSObject behaves like it contains this method:

```
def setValue_forKey(value, key)
  self.send(key+'=', value)
end
```

valueForKey and setValue_forKey are the heart of what Cocoa calls *key-value coding* (KVC). A class that's *KVC-compliant* must respond to valueForKey. If the property has any setters, one of them must be setValue_forKey. There are additional requirements if a setter does validation. I refer you to *Key-Value Coding Programming Guide* [App08n].

There's a subtlety here. Setting an observed property notifies any observers. So, even though our BoundObject was the object that changed the Root's property, that change is reflected right back to it, causing observeValueForKeyPath_ofObject_change_context to be called. Since that method doesn't change anything in the rooted keypath, the redundant update is harmless.[4]

You can run key-value/howtos/kv-binding-by-hand.rb to see the script in action:[5]

```
$ ruby kv-binding-by-hand.rb
```

27.7 In Summary...

Cocoa bindings are a three-layered technology.

Key-Value Coding

At the bottom, you have key-value coding, which is a way of making object properties more dynamic. Without it, the names of properties are hard-coded:

```
... root.property ...
```

4. If more than one rooted keypath could be bound to the same object, you'd want a change reflected up from one to be reflected down to the others, but that's not the way bindings are usually used. For example, there's no way in Interface Builder to set up such a configuration.

5. The script comments out lines that print how observeValueForKeyPath_ofObject_change_context is called. If you uncomment them, you may see that it's called more than once when upcase sets property's value. That's a bug in RubyCocoa 0.13.1. It has been fixed in later versions.

With it, it's easy to choose properties at runtime:

```
key = (be_formal ? 'last_name : 'first_name')
... root.valueForKey(key) ...
```

Key-value coding also lets you refer to a whole chain of object properties at once with the related method valueForKey*Path*:

```
code.valueForKeyPath("region.capital.name")
```

Some classes override key-value coding methods with useful behavior. For example, an NSArray overrides valueForKey to implement the collect-like behavior we've seen in NSArrayController:

```
class NSArray
  # ...
  def valueForKey(key)
    self.collect { | elt | elt.valueForKey(key) }
  end
end
```

The complete description of key-value coding is in Apple's *Key-Value Coding Programming Guide* [App08n].

Key-Value Observing

Key-value observing builds on top of key-value coding to allow objects to observe changes to other objects' properties. The properties being observed must be what Apple calls *KVO-compliant*. That means the setter calls willChangeValueForKey before a change and didChangeValueForKey after. Either of those methods will in turn call any observer's observeValueForKeyPath_ofObject_change_context method.

The complete description of key-value observing is in Apple's *Key-Value Observing Programming Guide* [App08m].

Cocoa Bindings

Bindings add no methods to Cocoa. They are a convention that Interface Builder encourages, one built on top of key-value observing. The convention is threefold:

- A binding between a binding object and a rooted keypath is established by a call to the binding object's bind_toObject_withKeyPath_options method.
- The binding object's observeValueForKeyPath_ofObject_change_context method changes state within the object, which is state that can be conceptualized with a single name (a noun or noun phrase).

- If the internal state that corresponds to the conceptual name changes, the value at the end of the rooted keypath is changed to match.

bind_toObject_withKeyPath_options takes an *options* argument. Cocoa provides a large set of default option names that assume that the bound object is a user interface control. For example, a number of them govern the way the object's selection is handled. These options are sketchily described in Apple's *NSKeyValueBindingCreation Protocol Reference* [App08r], which is part of the Cocoa API. Your own classes can ignore any of those options, or you can add options of your own.

You can find a list of the bindable controls, and which options they obey, in Apple's *Cocoa Bindings Reference* [App08g]. Apple's *Cocoa Bindings Programming Topics* [App08f] gives a deeper explanation of how bindings work with user interface controls.

27.8 Postscript: Observing Changes to Collections

Key-value observing can also be used with arrays (as well as a class we haven't used, NSSet). Consider this class:

Download key-value/tests/kv-observing-collections-tests.rb

```
class ArrayHolder < OSX::NSObject
❶  kvc_array_accessor :values

  def initWithValues(*initial_values)
    @values = NSMutableArray.arrayWithArray(initial_values)
    self
  end

❷  def values; @values; end
end
```

kvc_array_accessor (at ❶) notes that the property values is what the Cocoa documentation calls a *to-many relationship*. Note (at ❷) that it does not define the usual getter and setter methods. That's because the array is manipulated in different ways, as you can see in this test:

Download key-value/tests/kv-observing-collections-tests.rb

```
def setup
  @watcher = rubycocoa_flexmock
❶  @observed = ArrayHolder.alloc.initWithValues('old_at_0', 'old_at_1')
end
```

```
context "insert methods" do
  should "trigger observation" do
    # NSKeyValueObservingOptionOld is meaningless in case of insertion.
    add_observer(:options => NSKeyValueObservingOptionNew)
    during {
❷     @observed.insertObject_inValuesAtIndex('new_at_0', 0)
    }.behold! {
❸     watcher_should_receive_change { | actual |
        actual[:kind] == NSKeyValueChangeInsertion &&
        same_indexes(actual[:indexes], [0]) &&
        actual[:new] == ['new_at_0']
      }
    }
❹   assert { @observed.values == ['new_at_0', 'old_at_0', 'old_at_1' ] }
  end
end
```

To insert a new value, you use a special method (at ❷). You do *not* use normal array manipulations. (They would not cause the observer to be notified.)

The results of the insertion are shown in the difference between ❶ and ❹.

You can see the changes reported to the observer at ❸:

- The :kind of the change is a special value that indicates insertion into a to-many relationship.
- Since there are methods that allow replacement of multiple elements at once, an NSIndexSet describes the :indexes that changed.
- The change also includes the :new elements, given in the same order as their indexes.
- Because only the :new elements are passed to the observer, not the whole array, it makes no sense to ask for the :old elements—there aren't any. If the NSKeyValueObservingOptionNew option is given when the observer is added, it's ignored.

Deletion and replacement of an element are handled similarly, except that the methods are, respectively, removeObjectFromValuesAtIndex and replaceObjectInValuesAtIndex_withObject. Look in key-value/tests/kv-observing-collections-tests.rb for examples.

Appendix A

Glossary

action, action method
: A name for a method that's called by a user interface element—for example, in response to a button click. Action methods take a single argument, the sender. Introduced on page 28.

adapter
: A class that improves on an interface that it hides. Introduced on page 141.

archive (v.)
: To save a set of in-memory objects to disk such that the original set can be reconstituted later. In Fenestra, this is done with the NSKeyedArchiver and NSKeyedUnarchiver classes. They have the same purpose as the pure-Ruby Marshall class. Introduced in Section 11.2, *Archiving*, on page 134.

assertion
: In a test, a claim that a boolean expression will evaluate true (or, sometimes, false). An assertion is different from an *expectation*. Introduced on page 206.

attribute
: In Ruby, attributes are instance variables made externally accessible with *getters* or *setters*. In Objective-C, the word *property* is usually used instead.

  In Cocoa, attributes are more usually values set using Interface Builder's Attributes *inspector*. They may not be accessible with getters or setters. Sometimes they are *bound names*.

binding name
: See *bound name*.

bindings, Cocoa bindings
: A way of synchronizing values belonging to two different objects. See *bound* for a short explanation. First used in Chapter 13, *Implementing a Preference Panel with Cocoa Bindings*, on page 162.

bound (adj.)
: For one object (the *bound object*) to be bound to a *rooted keypath*, it must be able to use *key-value observing* to see changes to the *property* at the end of the keypath. The bound object should reflect the changes in its *bound name*. Changes to the bound object may also cause a change to the observed property. Introduced on page 162.

bound name, binding name
: The string name of some piece of a *bound object's* internal state. Introduced on page 187.

bound object
: An object, usually some sort of control, whose *bound name* changes in synchrony with a *property* at the end of a *rooted keypath*. Introduced on page 187.

bridge metadata
: Static information about an Objective-C method. It supplements the information RubyCocoa can determine at runtime. Introduced on page 347.

bundle
: A bundle is a directory hierarchy with a particular structure that allows it to be treated as a unit. Applications are one kind of bundle. Introduced on page 32.

business logic
: See *model*.

Carbon
: An older API that serves the same purpose as Cocoa. Carbon is written for the C programming language rather than Objective-C. Introduced on page 264.

cell
: An object attached to a *control*. The cell handles some of the work a user thinks is performed by a control. For example, NSCell objects

display text or images. Introduced on page 77, but see also Section 17.1, *Cells*, on page 230.

combo box (NSComboBox)
: A combination of a drop-down list and a text box. Introduced on page 76.

content view
: A *view* that covers the entire space of a window. It contains all the other views. Introduced on page 43.

control (NSControl)
: A control is an object that can draw a representation on the screen, respond to user events, and send *action* messages to *controllers*. Buttons and text fields are typical controls. All controls are *views*. Introduced on page 74.

controller
: An object that is the *target* of user interface *actions*. A controller mediates between *view* objects and *model* objects. Introduced on page 44.

data object
: An object that contains nothing but *getters* and *setters*. Also called a *dumb object*. Introduced on page 268.

data source
: Some *controls* do not contain the data they display. Instead, they fetch it from a data source when they need it. The data source must obey a *protocol* that defines the methods the control may use. Introduced on page 143.

default button
: The button that's "clicked" when the Return key is pressed. In Cocoa, the default button is colored solid blue. Introduced on page 77.

defaults (user)
: See *user preferences*.

delegate (n.)
: An object to which other objects delegate work. In Cocoa, delegation is often used in place of inheritance. Introduced on page 23.

doc window, document window
: In Interface Builder, the Finder-like window that represents the contents of the *nib*. Introduced on page 41.

document-based application
: An application that lets you view and perhaps edit documents stored on disk. Often, you can work on multiple documents at once. Introduced on page 313.

document controller (NSDocumentController)
: In a *document-based application*, an object that creates and manages *document objects*. Introduced in Section 24.1, *The Major Players*, on page 314.

document object (NSDocument)
: In a *document-based application*, the in-memory object that corresponds to an on-disk document. Introduced in Section 24.1, *The Major Players*, on page 314.

domain-specific language, DSL
: A little language that makes a solution easy to express. The solution is then written in that language. Introduced on page 100.

DSL
: See *domain-specific language*.

duck typing
: In duck typing, an object is of an appropriate type if it responds to a needed set of messages. It doesn't matter what the class of the object is. Introduced on page 24.

dumb object
: See *data object*.

dummy, test dummy
: An object that must exist but is never actually used in a particular test. For example: an ignored argument in a method call. A kind of *test double*. Introduced on page 284.

expectation
: *Mock objects* are programmed with expectations about how certain methods of the object will be called during a test. At the end of the test, any expectations that have not been met will cause a test failure. Introduced on page 215.

fake
: A fake is a *test double* that has the same behavior as the real object but in a way that's more convenient for testing. Introduced on page 284.

feature envy
: A method has feature envy if it seems more interested in the methods of another class than in its own. (That is, it sends more messages to an instance variable than to self.) A method with feature envy should often be moved to the other class. Introduced on page 268.

fenestra, fenestration
: A fenestra or fenestration is a hole. Fenestration is also the act of creating the hole. Introduced on page 18.

first responder
: The object getting the first chance to handle keyboard and mouse events. See also *initial first responder*. Introduced on page 80.

focus
: The *control* that responds to keypresses and mouse events. See *responder*. Introduced on page 80.

formatter (NSFormatter)
: An object that mediates between a *control* and some *model* object. The formatter converts an object into text, and vice versa. It can also support error-handling and field-specific editing. Introduced in Section 13.3, *Formatters*, on page 172.

frame
: A rectangle that defines the position and size of a window or *view*. Introduced on page 145.

frame name
: The name of a window's *frame*. If a window has a named frame, its position and size are stored in *user preferences* between app invocations. Introduced on page 287.

getter
: A method that returns the value of an instance variable. In Ruby, these are often defined with attr_reader.

group (Xcode)
: In Xcode, a way of putting related files next to each other. Groups are independent of disk folders. Introduced on page 118.

help book
: A collection of pages reached from the Help menu and displayed by the help viewer. An application usually has exactly one help book. Introduced on page 301.

initial first responder
: The *responder* that's made the *first responder* when a window is first placed on-screen. Introduced on page 80.

inspector (Interface Builder)
: In Interface Builder, the tabbed window you use for changing a *nib* object's attributes, connections, class, and so forth. Introduced on page 41.

key equivalent
: A key that, when pressed, is the equivalent of a mouse click on a particular control (such as a menu item or button). Introduced on page 28.

keypath
: A period-separated list of NSDictionary keys or class *property* names. Introduced on page 163.

key-value coding, KVC, KVC-compliant
: A *property* uses key-value coding if it supports the method valueForKey. If the property is settable, it should also support setValue_forKey. Introduced on page 357.

key-value observable, KVO, KVO-compliant
: An object's *property* is KVO-compliant if it either implicitly or explicitly uses willChangeValueForKey and didChangeValueForKey to signal when its value changes. Introduced on page 352 and on page 358.

key view loop
: The sequence of *controls* that the Tab key takes you through. Introduced on page 288.

key window
: The window that keypresses go to. Introduced on page 157.

lazy getter
: A *getter* for an object's *attribute* (or *property*) that calculates the value the first time it's called. Introduced on page 324.

library (Interface Builder)
: In Interface Builder, a collection of predefined user interface elements. You create items in the *doc window* or *main window* by dragging them out of the library and dropping them. Introduced on page 41.

loading (a nib file)
: See *nib loading*.

main menu
: The menu that appears in the menu bar on top of the screen. It is defined in a *nib file* that's usually named MainMenu.nib. Introduced on page 40.

main window
: The primary window created by Interface Builder and stored in a *nib file*. Introduced on page 41.

matrix (NSMatrix)
: A matrix acts to group *controls*. In some cases, as with radio buttons, it coordinates their behavior. Introduced on page 290.

mock object, mock
: A *test double* programmed with *expectations* of what sort of messages it will receive. The test fails if some of the expectations aren't met or, sometimes, if the mock gets messages it didn't expect. Introduced on page 284.

model, model object
: The model is a layer of the app whose job is to represent the app's conceptual world. For example, in an accounting package, the model would contain classes representing Accounts, Payees, Invoices, and the like. A model object is an instance of one of those classes. Introduced on page 223.

model-view-controller, MVC
: The name for a broad class of solutions to the problem of separating concerns in a GUI-heavy app. All the variants use three objects. The *view* handles the screen and fields user input, the *model* is concerned with data and *business logic*, and the *controller* coordinates between the two.

nib, nib file
: A file produced by Interface Builder that contains *archived* objects

and instructions for connecting *targets*, *outlets*, and *actions*. Nib files end in .nib or .xib. Introduced on page 40.

nib loading
: The process of reconstituting all of the *archived* objects in a nib file and connecting their *outlets*. Discussed throughout the book.

notifications
: A mechanism for decoupling classes. Instead of sending a message to one or more known objects, an object makes an announcement via an NSNotification object. Objects that register for that kind of notification receive it. Introduced in Chapter 7, *Notifications Connect Decoupled Objects*, on page 94.

observed property key, observed key, observable property key
: A *property* name or NSDictionary key that is *key-value observable*. Introduced on page 187.

observed property value, observed value
: The value associated with an *observed property key*. Introduced on page 187.

outlet
: An instance variable of a *nib* object that is set pointing to another *nib* object during *nib loading*. Introduced on page 44.

pasteboard (NSPasteboard)
: Pasteboard objects are places to stash data (or promises of data) that can be accessed by other apps. Cut and Copy use pasteboards, as does dragging and dropping. Introduced on page 262.

pointer (Objective-C)
: A pointer is a memory word containing a memory address (rather than data). Introduced on page 172.

preferences
: See *user preferences*.

property
: The Objective-C name for what Ruby calls an *attribute*: an instance variable made accessible through *getters* or *setters*. Introduced in a footnote on page 185.

protocol (Objective-C)
: A named group of related methods. For example, a class that

needs to be an NSComboBox *data source* must follow the NSComboBoxDataSource protocol. Introduced on page 262.

refactoring
: Changing code without changing its externally visible behavior. Introduced on page 283.

reopen (a class)
: To add new methods to an already defined class. Introduced on page 73.

responder (NSResponder)
: Any object that can handle keyboard and mouse events. Introduced on page 80.

responder chain
: The responder chain begins with the *first responder*. All responders may either handle or decline to handle an event. If they decline it, they indicate another responder to try. That sequence of responders is the responder chain. Introduced on page 80. See also discussion on page 317.

rooted keypath
: A rooted keypath begins at a root object. The *keypath* is made up of one or more names of *KVC-compliant properties* (ones that can be queried with valueForKey). The first property in the keypath leads from the root to another object. The next property leads from that object to yet another object, and so on. Introduced on page 187.

selection
: A marked subset of the contents of a control. Some user gestures (such as copy) will apply only to the selection. In Fenestra, selections come into play most strongly because some table operations require a row to be selected. Introduced on page 202.

selector
: The name of an Objective-C method. Introduced in a footnote on page 56.

setter
: A method that changes the value of an instance variable. In Ruby, these are often defined with attr_writer.

spike, spike solution
: A coding exercise, done in response to a problem, that helps you

understand how to solve the problem. Spikes are usually throw-away code. Introduced on page 314.

struct
: A struct is an Objective-C construct. Like an object, it contains data, but it has no methods (not even *getters* or *setters*). In RubyCocoa, structs have to be represented by special objects, such as instances of NSRange, that are converted to structs when passed into Objective-C. Introduced on page 54.

stub (test)
: A *test double* that gives the same answers as the real object would but only for those questions a test asks. It can't handle everything the real world might throw at it. Introduced on page 284.

target, target object
: An object designated as the recipient of an *action*. Introduced on page 28.

text attributes
: Characteristics of runs of characters inside an NSAttributedString. One example: a font that applies to characters in the range given by NSRange.new(3, 34). Introduced on page 321.

text field (NSTextField)
: A *control* that displays a line of text the user can edit or select. A test field sends its *action* message to its *target* when `Return` is pressed. In Cocoa, labels are uneditable, unselectable text fields. Introduced on page 37.

test double
: An object that replaces a real object for testing purposes. Test doubles can be *dummies*, *stubs*, *mocks*, or *fakes*. Introduced on page 283.

text view (NSTextView)
: Text laid out in a rectangular area. A text view is larger than a *text field* and allows more sophisticated editing. Introduced on page 37.

toggle button
: A button that changes state from NSOffState to NSOnState. Typically, the text of the button changes to match the state. Introduced on page 77.

to-many relationship
: When the value of a *property* is a collection object (for example, NSArray or NSSet), the property is said to have a *to-many relationship* to the elements of the collection. *Key-value observers* can be notified of element insertions, deletions, and replacements. Introduced in Section 27.8, *Postscript: Observing Changes to Collections*, on page 359.

translator (Fenestra)
: In Fenestra, an object that changes between-process *notifications* into strings. Introduced on page 76.

user preferences, user defaults
: A system that makes it convenient to store key/value pairs on disk. Most usually, the values are window positions the user prefers, default folders, and the like. Introduced in Section 11.1, *The User Preferences System*, on page 128.

value transformer (NSValueTransformer)
: An object that mediates between a *bound object* and a *rooted keypath*. It transforms values flowing from one to the other. Introduced on page 177.

view (NSView)
: A view is an area within a window or another view that has its own coordinate system. It can handle events that pertain to its *frame*, and it can draw to that frame. All *controls* are views. Introduced in Section 3.1, *Big views have little views...*, on page 42.

virtual machine, VM
: A software substrate that aids in the implementation of languages like Ruby or Objective-C. Introduced on page 334.

window controller (NSWindowController)
: In a *document-based application*, the *controller* controlling the window that corresponds to a *document object*. Introduced in Section 24.1, *The Major Players*, on page 314.

Appendix B

Bibliography

[App08a] Apple, Inc. *Apple Help Programming Guide*. http://developer.apple.com/documentation/Carbon/Conceptual/ProvidingUserAssitAppleHelp, 2008.

[App08b] Apple, Inc. *Apple Human Interfaces Guidelines*. http://developer.apple.com/documentation/userexperience/Conceptual/AppleHIGuidelines, 2008.

[App08c] Apple, Inc. *Archives and Serializations Programming Guide for Cocoa*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/Archiving, 2008.

[App08d] Apple, Inc. *Bundle Programming Guide*. http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles, 2008.

[App08e] Apple, Inc. *Button Programming Topics for Cocoa*. http://developer.apple.com/documentation/Cocoa/Conceptual/Button, 2008.

[App08f] Apple, Inc. *Cocoa Bindings Programming Topics*. http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaBindings, 2008.

[App08g] Apple, Inc. *Cocoa Bindings Reference*. http://developer.apple.com/documentation/Cocoa/Reference/CocoaBindingsRef, 2008.

[App08h] Apple, Inc. *Cocoa Event-Handling Guide*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/EventOverview, 2008.

[App08i] Apple, Inc. *Combo Box Programming Topics*. http://developer.apple.com/documentation/Cocoa/Conceptual/ComboBox, 2008.

[App08j] Apple, Inc. *Document-Based Applications Overview*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/Documents, 2008.

[App08k] Apple, Inc. *Drag and Drop Programming Topics for Cocoa*. http://developer.apple.com/documentation/Cocoa/Conceptual/DragandDrop, 2008.

[App08l] Apple, Inc. *Internationalization Programming Topics*. http://developer.apple.com/documentation/MacOSX/Conceptual/BPInternational, 2008.

[App08m] Apple, Inc. *Key-Value Coding Observing Guide*. http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueObserving, 2008.

[App08n] Apple, Inc. *Key-Value Coding Programming Guide*. http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding, 2008.

[App08o] Apple, Inc. *Launch Services Programming Guide*. http://developer.apple.com/DOCUMENTATION/Carbon/Conceptual/LaunchServicesConcepts, 2008.

[App08p] Apple, Inc. *Matrix Programming Guide for Cocoa*. http://developer.apple.com/documentation/Cocoa/Conceptual/Matrix, 2008.

[App08q] Apple, Inc. *Notification Programming Topics for Cocoa*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/Notifications, 2008.

[App08r] Apple, Inc. *NSKeyValueBindingCreation Protocol Reference*. http://developer.apple.com/documentation/Cocoa/Reference/ApplicationKit/Protocols/NSKeyValueBindingCreation_Protocol, 2008.

[App08s] Apple, Inc. *Objective-C 2.0 Runtime Programming Guide*. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjCRuntimeGuide, 2008.

[App08t] Apple, Inc. *Pasteboard Programming Topics for Cocoa*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/CopyandPaste, 2008.

[App08u] Apple, Inc. *Property List Programming Guide*. http://developer.apple.com/documentation/Cocoa/Conceptual/PropertyLists, 2008.

[App08v] Apple, Inc. *Resource Programming Guide*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/LoadingResources, 2008.

[App08w] Apple, Inc. *Ruby and Python Programming Topics for Mac OS X*. http://developer.apple.com/documentation/Cocoa/Conceptual/RubyPythonCocoa, 2008.

[App08x] Apple, Inc. *Runtime Configuration Guidelines*. http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig, 2008.

[App08y] Apple, Inc. *Text System Overview*. http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/TextArchitecture, 2008.

[App08z] Apple, Inc. *Text System User Interface Layer Programming Guide for Cocoa*. http://developer.apple.com/documentation/Cocoa/Conceptual/TextUILayer, 2008.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, second edition, 1996.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[Ful06] Hal Fulton. *The Ruby Way*. Addison-Wesley, Reading, MA, second edition, 2006.

[Mar06] Brian Marick. *Everyday Scripting with Ruby: For Teams, Testers, and You*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Reading, MA, 2007.

[Ols07] Russ Olsen. *Design Patterns in Ruby*. Addison-Wesley, Reading, MA, 2007.

[Rai05] J. B. Rainsberger. Injecting testability into your designs. *Better Software*, 2005.

[TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

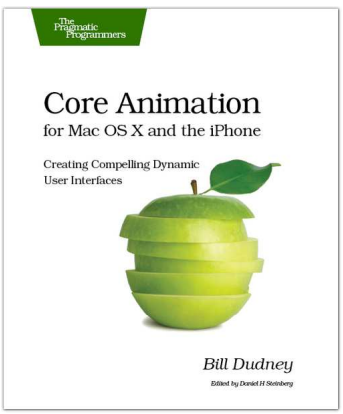
W

X

More on Cocoa and Ruby

Core Animation for OS X/iPhone

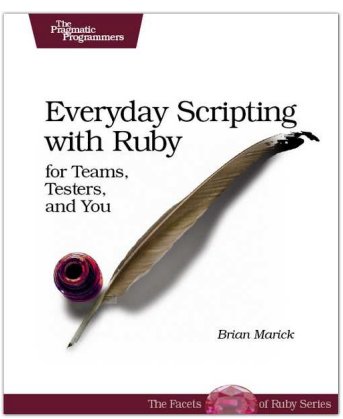
Have you seen Apple's Front Row application and Cover Flow effects? Then you've seen Core Animation at work. It's about making applications that give strong visual feedback through movement and morphing, rather than repainting panels. This comprehensive guide will get you up to speed quickly and take you into the depths of this new technology.



Core Animation for Mac OS X and the iPhone: Creating Compelling Dynamic User Interfaces
Bill Dudney
(220 pages) ISBN: 978-1-9343561-0-4. $34.95
http://pragprog.com/titles/bdcora

Everyday Scripting with Ruby

Don't waste that computer on your desk. Offload your daily drudgery to where it belongs, and free yourself to do what you should be doing: thinking. All you need is a scripting language (free!), this book (cheap!), and the dedication to work through the examples and exercises. Learn the basics of the Ruby scripting language and see how to create scripts in a steady, controlled way using test-driven design.



Everyday Scripting with Ruby: For Teams, Testers, and You
Brian Marick
(320 pages) ISBN: 0-9776166-1-4. $29.95
http://pragprog.com/titles/bmsft

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Cocoa with Ruby's Home Page
http://pragprog.com/titles/bmrc
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/bmrc.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)