Using Bindings with Pop-Up Menus

Author: Mark Szymczyk
Last Update: December 8, 2005

Bindings are a Cocoa technology that simplifies the process of keeping data and user interface elements in sync. Most articles on bindings use a table view as an example. This article shows you how to use bindings with a pop-up menu.

Introduction

A bedrock of object-oriented programming is the Model-View-Controller (MVC) design pattern. Model objects store data. View objects are user interface elements. Controller objects keep the model and view objects synchronized. Before bindings you had to write a lot of code for controller objects. Apple introduced bindings to Cocoa in Mac OS X 10.3 to reduce the amount of code you have to write.

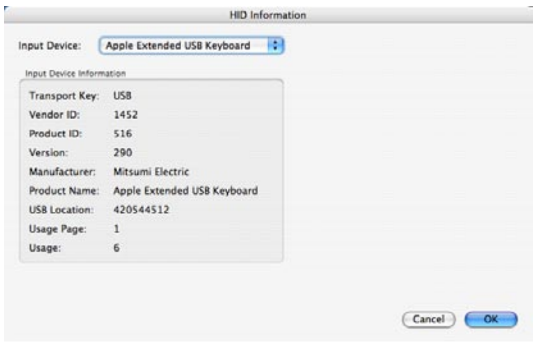
The bindings technology has two components. The first component is a series of controller classes that manage one or more model objects. There are five controller classes, which you can find in Interface Builder's Controllers Palette.

- Object controllers manage one object.
- Array controllers manage multiple objects.
- User defaults controllers manage user preferences.
- Managed object contexts work with Cocoa's Core Data framework, which lets you create model objects without writing code.
- Tree controllers manage a tree of objects.

The second component of the bindings technology consists of the bindings themselves. Each user interface element has its own set of bindings, which you can see in Interface Builder by selecting an element and pressing Cmd-4. A binding connects one aspect of the user interface element with a controller or model object. The binding connects the view and the data.

The Example Program

I'm going to make a simplified version of the HID Explorer program that is part of Apple's sample code collection. The program shows a dialog box consisting of a pop-up menu and text fields. The pop-up menu contains the names of the HID devices connected to the user's Mac. Selecting a device from the menu fills the text fields with the HID Manager data about the device.



Input Device: Apple Extended USB Keyboard

Transport Key:	USB
Vendor ID:	1452
Product ID:	516
Version:	290
Manufacturer:	Mitsumi Electric
Product Name:	Apple Extended USB Keyboard
USB Location:	420544512
Usage Page:	1
Usage:	6

There are two main classes for this program: `InputController` and `InputDevice`. The `InputController` class manages the input devices connected to the user's machine.

```
@interface InputController : NSObject {
    NSMutableArrayPtr inputDeviceList;
    NSObject* currentlySelectedItem;
}
```

The inputDeviceList member contains the HID devices connected to the user's Mac. The `currentlySelectedItem` member holds the device the user chooses from the pop-up menu. To get the bindings to work for a pop-up menu, you need a data member like `currentlySelectedItem` someplace in your program. If you're using Core Data, a common technique is to add a variable of type `NSManagedObject` to the `AppDelegate` class.

The `InputDevice` class contains the HID data for an input device.

```
@interface InputDevice : NSObject {
    io_object_t deviceObject;
    NSStringPtr transportKey;
    long vendorID;
    long productID;
    long version;
    NSStringPtr manufacturer;
    NSStringPtr productName;
    NSStringPtr serialNumber;
    long usbLocationID;
    long usagePage;
    long usage;
}
```

The important pieces of data are `productName`, `usagePage`, and `usage`. As you could probably tell, `productName` is the name of the device. The combination of `usagePage` and `usage` describes what the device is: keyboard, mouse, joystick, gamepad, or steering wheel.

I've included a finished project you can download. If you want to work through the example instead of using the finished project,

1) Create a Cocoa application project.
2) Add the IOKit framework to the project.
3) Add the files `InputController.h`, `InputController.m`, `InputDevice.h`, and `InputDevice.m` to the project.
4) Open the file `MainMenu.nib` in Interface Builder.
5) Add a pop-up button control, static text items, and two buttons to the window to match the screenshot on Page 1. Leave the text fields on the right side (USB, 1452, 516, 290, etc.) blank. The bindings are going to fill those values when you run the program.
6) Instantiate `InputController` and `InputDevice`. Either drag the header files from Xcode to the nib file window in Interface Builder or choose Classes > Read Files in Interface Builder.

Create the Controller

The first step to using bindings is to create a controller object. To create a controller object, select a controller from Interface Builder's Controllers Palette and drag it to the nib file window. If you're working through this example, drag an array controller to the nib file window. I named my array controller `InputDeviceMenuController`. You can give your controller a different name if you want.

After creating the array controller you must specify what the array controller is going to manage. For this example the array controller is going to manage the HID devices connected to the user's computer. The `InputController` class stores the device list so you must make a connection from the array controller to the `InputController` class. In Interface Builder select the array controller and control-drag to the `InputController` instance in the nib file window. The information panel will show the list of outlets for the array controller. Select the `content` outlet and click the Connect button.

At this point you've connected the controller to the object holding the array. Now you must tell the controller what types of objects are in the array. Select the array controller from the nib file and press Cmd-1. Set the object class name to `InputDevice`, which tells the array controller it's managing an array of `InputDevice` objects.

At the bottom of the information panel you will see a list of keys. The controller uses keys to bind data to user interface elements. Click the Add button to add a key to the list. Add the following keys: `manufacturer`, `productID`, `productName`, `serialNumber`, `transportKey`, `usage`, `usagePage`, `usbLocationID`, `vendorID`, and `version`.

Bind the Pop-up Menu

Now it's time to bind the user interface elements. Choose Bindings from the menu at the top of the information panel or press Cmd-4. The first user interface element to bind is the pop-up menu. There are three bindings you must set for the pop-up menu. The `content` binding describes what the collection of objects is. In this example the collection of objects is the objects in the array controller. In Interface Builder, click the disclosure triangle next to the `content` binding and make the following choices:

```
Bind to: InputDeviceMenuController
Controller Key: arrangedObjects
Model Key Path: Leave Blank
```

The `contentValues` binding describes what will be displayed in the pop-up menu. In this example it will be `productName`, the name of the input device. Make the following choices for the `contentValues` binding:

```
Bind to: InputDeviceMenuController
Controller Key: arrangedObjects
Model Key Path: productName
```

The `selectedObject` binding describes the object the user chose from the menu. For a table view you could use the array controller with the controller key `selection`. But pop-up menus don't record the selection the user made so the `selection` controller key doesn't work. That is why you need the `currentlySelectedItem` data member of the `InputController` class. You're going to bind the `selectedObject` binding to `currentlySelectedItem`. Make the following choices for the `selectedObject` binding:

```
Bind to: InputController
Controller Key: Leave Blank
Model Key Path: currentlySelectedItem
```

Bind the Text Fields

Binding the text fields is going to be easier than binding the pop-up menu. There is only one binding you must set for each text field. The `value` binding specifies what will be displayed in the text field. What is going to be displayed is one of the fields of the `currentlySelectedItem` variable. For the text field next to the Transport Key label, make the following choices for the `value` binding:

```
Bind to: InputController
Controller Key: Leave Blank
Model Key Path: currentlySelectedItem.transportKey
```

Binding the other text fields will be similar. The Model Key Path is the only field that will change.

Finishing the Application

If you followed along by creating your own version of the project, you could build and run the project from Xcode. The window would open and when you chose an input device from the pop-up menu, the text fields would contain the HID information about the device. But the OK and Cancel buttons wouldn't work. There's a little more work to be done to finish the application.

Make the OK and Cancel Buttons Work

When the user clicks the OK or Cancel buttons, the window should close. To get this behavior,

1) Make connections from each button to the window. Control-click each button and drag to the window's instance in the nib file window.
2) Click the Target/Action tab in the information panel.
3) Select `orderOut:` from the lists of actions and click the Connect button.

When you selected `orderOut:`, you probably noticed the `performClose:` action. Since the window should close when the user clicks the OK or Cancel buttons, `performClose:` sounds like the action to perform. Why choose the `orderOut:` action? I don't know why, but using the `performClose:` action doesn't cause the window to close when the user clicks the OK or Cancel buttons. The `orderOut:` action hides the window, which lets you easily show the window if the user wants to see the window again.

You also should assign keyboard equivalents for the OK and Cancel buttons: the Return key for OK and the Esc key for cancel. Select each button and press Cmd-1. Choose the appropriate key from the Key Equivalent menu.

Make the Window Invisible Initally

I chose to have the example program's window open when the user chooses File > View HID Information. To get this behavior the window must be invisible initially. Click inside the window, but not on any of the window's user interface elements. Press Cmd-1 and deselect the Visible at Launch Time checkbox.

Modify the Menus

The example program is simple. It requires only one menu item. Remove the Edit, Window, and Help menus from the menu bar. Remove all of the menu items from the File menu except one. Double-click that item and rename it View HID Information.

When the user chooses File > View HID Information, the window should open. To get this behavior, perform the following steps:

1) Make a connection from the View HID Information menu item to the window. Control-click the menu item and drag to the window's instance in the nib file window.
2) Click the Target/Action tab in the information panel.
3) Select `makeKeyAndOrderFront:` from the list of actions and click the Connect button.

Conclusion

This article only scratched the surface of Cocoa bindings. To learn more about bindings, read the document *Cocoa Bindings Programming Topics*. This document is part of Apple's developer documentation, which you can read in Xcode or at Apple's developer site. The document is in the Cocoa section under Design Guidelines.

Scott Stevenson wrote a 14 page article for CocoaDevCentral that provides a gentle introduction to bindings. The combination of his article and this article should provide enough of a background so you can start using bindings in your Cocoa applications.