Technical Note 2005
SimpleCocoaApp: An Overview

CONTENTS

This Technote takes a look at the SimpleCocoaApp code sample and uses it as a jumping off point for discussing Cocoa programming tips and principles. You are encouraged to download the SimpleCocoaApp sample from here and study the source code directly, in conjunction with this Technote. While not everything that SimpleCocoaApp does would be found in every application, all of it has applications somewhere.

While this technote is geared towards the relative beginner at Cocoa, it does assume some basic knowledge of Objective-C, Project Builder, and Interface Builder. Links to additional Cocoa documentation are listed at the end of this technote. The goal here is to help open your horizons by simply demonstrating some of the power of the application development system and APIs. Hopefully, a few previously fuzzy concepts will be made more clear, and you will be inspired to explore the world of Cocoa.

Updated: [Jan 21 2002]

What Is SimpleCocoaApp?

SimpleCocoaApp is a really simple application. No, really. SimpleCocoaApp consists of less than 20 lines of code, but that is all it takes to get a whole truckload of features. This application admittedly doesn't do very much useful work, but it does handle radio buttons, Pop-Up menus, the Help menu, window processing, button clicks, text field input, and an About Box. Here's a screen shot of the application in action (see figure 1):

Figure 1 Shows the main window of SimpleCocoaApp.

SimpleCocoaApp uses three main objects, instantiated (created) from three different custom classes:

- Hello1: Contains implementations of the methods `-message1:`, `-message2:`, and `-message3:`
- Hello2: Contains different implementations of `-message1:`, `-message2:`, and `-message3:`
- HelloController: Handles the data interaction between the GUI and the Hello objects, as well as displaying the ReadMe help file in response to a "SimpleCocoaApp Help" selection from the Help menu. It's usually good to have a "controller" object to manage the interaction between your view (GUI) and your model (the actual data objects; Hello objects in this case). This is known as the model-view-controller design paradigm.

Here's how the different objects interact:

Most controls in Cocoa have a *target* object associated with them, which is sent an *action message* (a message is similar to a method call) when the control is triggered. How the control looks on the screen and functions as a widget is handled by the Cocoa infrastructure, so that all you have to implement are the target objects and their action messages. Such is the case here.

- When one of the radio buttons is selected, an action message is sent automatically to the radio button's target (HelloController), which then changes the action message for the "Say Hello..." button to `-message1:`, `-message2:`, or `-message3:`.
- When a menu item in the pop-up menu is selected, an action message is sent to the pop-up menu's target (also HelloController), which then changes the "Say Hello..." button's target to a different "Hello" object (Hello1 or Hello2).
- When you click the "Say Hello..." button, it sends an action message (chosen by the radio buttons) to its target (chosen by the pop-up menu), which simply displays an alert (using the text in the "My Message..." text field if message #3 has been chosen).

Now let's explore the process of writing this application.

Back to top

Setting Things Up in Interface Builder

After creating a new Cocoa application in Project Builder, typically the first thing you will do is open MainMenu.nib to design your user interface and classes in Interface Builder. Since SimpleCocoaApp has already been written for you, you can simply double-click the Project Builder .pbproj file to open the project and then double-click MainMenu.nib in the left-hand-side file list. In MainMenu.nib, you will find the Hello1, Hello2, and HelloController classes under the Classes tab in the nib window, and the instances of Hello1, Hello2, HelloController and MyWindow under the Instances tab. The first step taken in SimpleCocoaApp was to create the Hello1, Hello2, and HelloController custom classes (by hitting return while a class like NSObject was selected under the Classes tab). Further steps included instantiating instances of these custom classes, and dragging controls/views into place on the window (MyWindow in this case). Apple encourages developers to adopt Aqua user interface guidelines for a consistent user experience; a link to further documentation can be found at the end of this technote. Once you've reached this stage in *your* application, you will want to connect up UI and your custom classes so they can send each other messages and tell each other to do things (grab values, set values, change behavior, whatever). This is done **graphically here in Interface Builder, so there is often no need to do any of it in code.** Instantiated objects are connected in two main ways in Interface Builder:

1. Any object (and controls in particular) can have an *outlet*, which is an instance variable pointing to some other object. Often it is useful to have your program's controller object have outlets to various user interface elements, so that it can tell them to change their state, etc.
2. Controls have a special way they can connect to other objects: they can assign an object to be the *target* of the control, and a particular method inside that object to be the *action* that gets called when the control is triggered.

To establish either of these connections, you control-click on the object that will be the source of messages, and drag the cursor to the object that will be the destination of the messages. This creates a gray line between the two objects. Then you can navigate the outlets and actions in the inspector, and click "Connect" to establish the connection. You will find that establishing outlet connections involves dragging one way, and establishing target-action connections involves dragging back the other way between objects. The key thing to remember is to *always control-drag in the direction that messages will flow.*

For example, in Figure 2 you can see that we are connecting an outlet called "messageTextField" in the instantiated Hello1 object to the NSTextField object that will hold the text to display for Hello1's `-message3:` method. We do this by control-dragging *from* the Hello1 object *to* the NSTextField. This is because when the program executes, the Hello1 object will send messages to the text field.

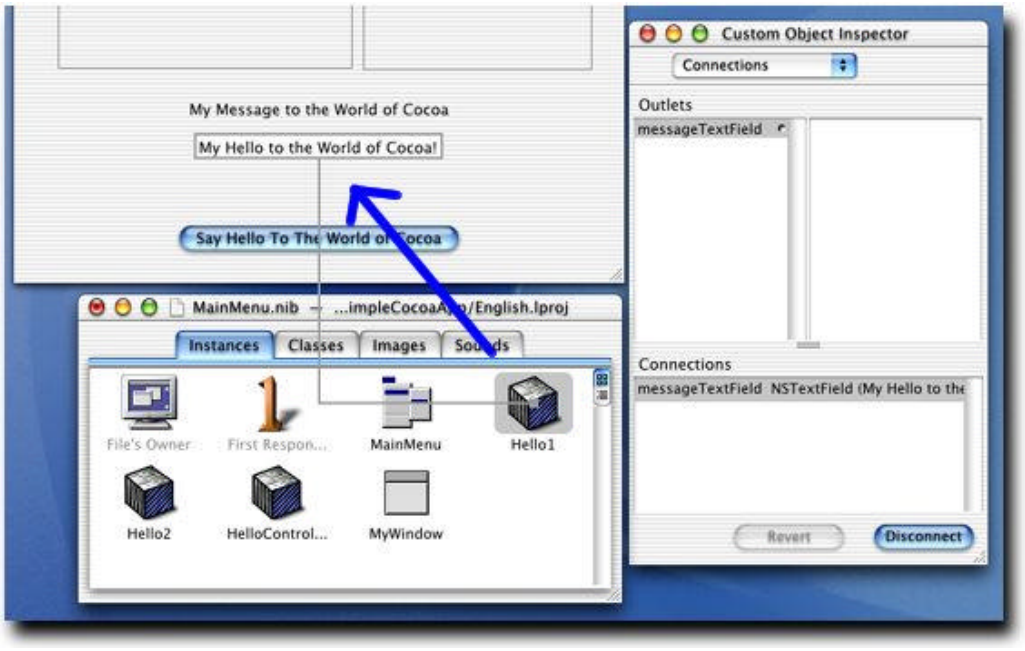

Figure 2. shows the establishment of an outlet connection between Hello1 and the NSTextField.

On the other hand, Figure 3 shows us setting up the target-action connection for the "Say Hello..." button. Since the button will send the action message to its target when the button is clicked, we can see that messages flow from button to target, and thus we control-drag from the button to Hello1. Then we choose the action "message1:" and click "Connect" in the inspector.

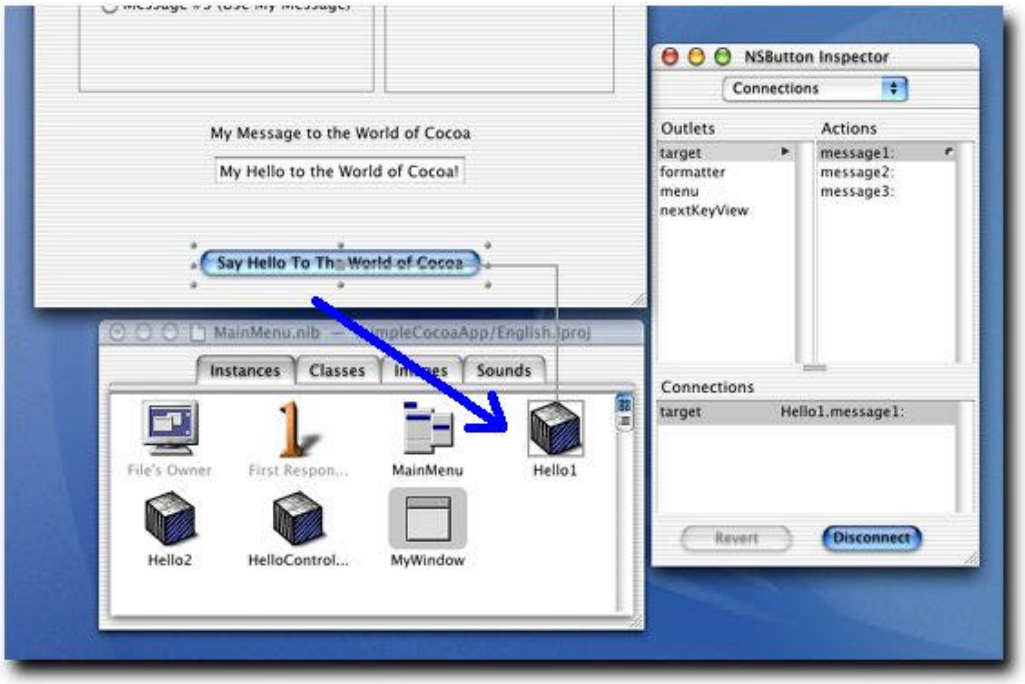

Figure 3. shows the establishment of a target-action connection between the "Say Hello..." button and Hello1.

Though it takes time to graphically setup the user interface, in the long run it saves time over trying to accomplish the same thing in pure source code. A this point, hitting Command-R (Test Interface in the File menu) will run a great deal of the user-interface portion of the application, without any code. Controls will work, buttons will click (though no actions wired to your custom code objects will be called), menus will appear, and so on. **All this is the functionality you get for free in Cocoa without writing one line of code!**

The final step to take before turning to the source code is to create skeleton header/source files for your custom classes by control-clicking on them in the "Classes" tab and choosing "Create Files...". Then, with the user interface designed, the custom objects instantiated, and the connections all made, we can turn to the source code for SimpleCocoaApp that implements our custom functionality.

Back to top

Main.m

```
int main(int argc, const char *argv[]) {
    return NSApplicationMain(argc, argv);
}
```

Listing 1. short and sweet!

As can be seen in Listing 1, nothing much goes in `main()` - you generally will just leave it like so. It simply jump-starts NSApplication, which runs the rest of the application, event loop and all. It is worth pointing out here that while all the code for SimpleCocoaApp has been written using Objective-C, you can also use Java to write your Cocoa applications - most of the objects/method names are the same.

Back to top

The Hello1 and Hello2 Objects

The Hello1 and Hello2 classes are virtually identical, the only difference being in the identifying title string placed on the dialog they present. The description here of Hello1 applies equally to Hello2 - the point of having the two classes is to demonstrate how to switch a control's target on the fly, so that only at runtime can you know for sure what lines of code will be executed when the "Say Hello..." button is pressed.

We turn now to Hello1.m, the implementation file for the Hello1 class. Although a class skeleton with empty methods can be easily generated by Interface Builder, it is up to the programmer to write the method implementations. Hello1 has three methods to implement: `-message1:`, `-message2:`, and `-message3:`.

```
- (IBAction)message1:(id)sender{
    //Use a standard alert to display the message
    NSRunAlertPanel(@"Message1, Hello1",@"Hello, Cocoa!",
        @"OK",NULL,NULL);
}
```

Listing 2. -message1: and -message2: are both implemented like so.

`NSRunAlertPanel()` (see Listing 2 above) presents the user with an alert box, and its parameters enable us to configure it. The first parameter is the alert title, the second is the alert message, and the next three represent up to three buttons in the alert. Passing NULL to a button parameter removes that button from the alert. Though we do not care about a return value from `NSRunAlertPanel()` here, it is important to note that it does indeed return an integer (for which there are predefined constants) representing which button was pressed.

```
- (IBAction)message3:(id)sender
{
    //Use a standard alert to display the text
    //in the messageTextField NSTextField
    NSRunAlertPanel(@"Message3, Hello1",
        [messageTextField stringValue],@"OK",NULL,NULL);
```

}

Listing 3. -message3: is a little more clever than the other two message routines.

In Listing 3 above, `-message3:` doesn't have `NSRunAlertPanel()` display a hard-coded line of text, but instead the contents of the NSTextField that the `messageTextField` instance variable points to. As part of the SimpleCocoaApp design, we connected the `messageTextField` outlet to the NSTextField control in Interface Builder, so that we could pull the value of the NSTextField like we are doing here. The `-stringValue` method returns the contents of an NSTextField as an NSString.

Back to top

The HelloController Object

HelloController is the real "brains" class behind SimpleCocoaApp. It does whatever processing is needed when the app starts up, it handles switching action messages and targets, and makes sure that help is displayed properly. It does all this in two short methods.

```
- (IBAction)switchMessage:(id)sender
{
    //sender is the NSMatrix containing the radio buttons.
    //We ask the sender for which row (radio button) is
    //selected and add one to compensate for counting from zero.

    int which=[sender selectedRow]+1;

    //We now set our NSButton's action to be the message
    //corresponding to the radio button selection.
    //+[NSString stringWithFormat:...] is used to concatenate
    //"message" and the message number.
    //NSSelectorFromString converts the message name string
    //to an actual message structure that Objective-C can use.

    [helloButton setAction:NSSelectorFromString([NSString
        stringWithFormat:@"%@%d:",@"message",which])];
}
```

Listing 4. - switchMessage: is called when the user picks a different message to display by clicking a different radio button.

The really important line in Listing 4 above is the last one. `+stringWithFormat:` takes a `printf`-style format string and a parameter list, generating a string in a manner similar in concept to C's `sprintf` routine. We pass the result from that to the `NSSelectorFromString()` function, which as its name implies, converts a string to a "selector," a method's full name/identifier (i.e. "method:withParameter1:parameter2:parameter3:"). That selector is then passed to `helloButton`'s `-setAction` method, changing the action called by `helloButton` when it is clicked.

```
- (IBAction)switchObject:(id)sender
{
    //sender is the NSPopUpMenu containing Hello object choices.
    //We ask the sender for which menu item is selected and add one
    //to compensate for counting from zero.

    int which=[sender indexOfSelectedItem]+1;

    //Based on which menu item is selected, we set the target
    //(the receiving object)
    //of the helloButton to point to either hello1 or hello2.

    if (which==1)
    [helloButton setTarget:hello1];
    else
    [helloButton setTarget:hello2];
}
```

Listing 5. -switchObject: is called when the user picks a different object to receive messages using the PopUp menu.

`helloButton` is an outlet (wired up in Interface Builder) that points to the "Say Hello..." NSButton on the window. Here we invoke the setTarget method to change the button's target to either hello1 or hello2, depending upon which object was chosen in the pop-up menu. Thus, when the button is pressed in the future, the new target will be sent the action message.

So that is all the source code. But what is powering the Help menu for SimpleCocoaApp? Where is the code for that? It turns out that with Cocoa you don't need any code to do it! By putting your html help files in the Resources folder in the application bundle, setting the "AppleTitle" html metatag to list the title of your "help book," and tweaking your Info.plist file (that goes automatically in the app bundle and is editable from Project Builder), you can get automatic Help Menu support - Help Viewer will be launched at the appropriate time, and your help book (merely an html file titled

"ReadMe.html" in this case) will be opened.

Back to top

Summary

All in all, about 10 lines of source code - and look at all the functionality you get! The real challenge for you the developer going on from here is to learn as much of the Cocoa frameworks as possible, so that you don't reimplement something yourself which is already done for you by Cocoa. Below are some links to documentation and the sample code this technote is based on.

Back to top

Online Documentation

Mac OS X Developer Documentation (as well as Adopting Aqua documentation)

Cocoa Developer Documentation

Developer Tools

Objective-C Manual

Project Builder

Back to top

Downloadables

	Acrobat version of this Note (392K).	Download
	Stuffed SimpleCocoaApp sample code (size in bytes, size in K)	Download

Back to top
