17

Building a Text Editor Using Multiple Nib Files

Most larger applications benefit from storing different parts of their interface in separate nib files. The primary elements of the interfaceÐthe main menu and perhaps a window or twoÐare contained in one nib file, and the other parts of the interface are contained in one or more auxiliary nib files. When the application starts, its primary interface objects are created immediately. Objects specified in its auxiliary nib files are created only on demand, as when a user requests an Info panel.

This program design is a consequence of the way nib files are accessed by an application. As you've seen in the earlier projects, all objects described in a nib file are created at the same time:

```
[NXApp loadNibSection:"Interface.nib" owner:NXApp];
```

There's no way to load a subset of a nib file's objects. However, the same functionality can be gained by using multiple nib files.

Using multiple nib files can improve your application's perceived performance. If at start-up time, an application creates only those objects a user will need immediately, the time it takes to start the application can be reduced. Of course, when users attempt to access other parts of the application, they will experience small delays as new objects are created from the auxiliary nib files. However, these delays are minimal and are incurred only when a user requests a specific part of the interface, rather than being imposed indiscriminately on all users when the application starts.

An equally important reason to have more than one nib file is to let an application replicate a piece of its interface any number of times. The document windows in Edit provide a good example. Since it can't be predicted how many document windows a user might need, the application must offer a way to create an unlimited number of them. By putting the document window interface in a separate nib file, each time a user requests another window, a new set of objects can be created from the file.

This project demonstrates how to use multiple nib files in an application. Before tackling the more advanced problem of using auxiliary nib files to replicate a piece of an application's interface, let's see how to use such a nib file to store an infrequently accessed user-interface object, the Info panel.

Adding an Info Panel to Your Application

An Info panel is an important component of your application's user interface; however, in practice users rarely access it. By putting the Info panel in a separate nib file, you can reduce your application's start-up time and memory usage. Let's see how this is done.

Close any other projects you may still have open and then choose the New command from Project

Builder's Project menu. Save the new project in your home directory under the name ªTextEditorº. In Project Builder's Files display, locate the entry for the interface file **TextEditor.nib** and double-click it to start Interface Builder. When Interface Builder starts, the new application's main menu and standard window appear. For now, these two components will constitute the application's primary user interface.

Next, let's create a class, the Distributor class, that defines an object to manage the Info panel. A Distributor object will be the target of an action message from the Info menu item. When it receives the Info item's message, the Distributor object will load the Info panel's interface.

To create the Distributor class, switch to the Classes display of Interface Builder's File window and scroll to the left to reveal the Object class. Click the Object class so it's the only class that's selected in the browser. Now, create a subclass of Object by dragging to Subclass in the pull-down list. When you release the mouse button, a new class is inserted in the class hierarchy. Using the Class Inspector, change the name of this class to ªDistributorº.

The next step is to declare the Distributor class's single outlet and action method. Make sure the Outlets button in the Class Inspector is highlighted and then enter **infoPanel** in the text field. Click Add Outlet. Next, click the Actions button and then enter **showInfoPanel:** in the text field. Click Add Action. The Attributes display should now look like this:

Figure 17-1. Attributes Display for the Distributor Class

To create template source code files for the Distributor class, drag to Unparse in the File window's pull-down list. Two panels open in succession: The first asks you to confirm that you want to create these class files, and the second asks whether these class files should be added to the project. Click OK in each panel. If you look at the Files display in the Project Inspector, you'll notice that **Distributor.h** is listed under Headers and **Distributor.m** is listed under Classes. We'll defer writing the **showInfoPanel:** method until the nib file that contains the Info panel has been created.

Now, let's create an object of the Distributor class and make it the target of the Info command. With the Distributor class selected in the File window, drag to Instantiate in the pull-down list. When you release the mouse button, the File window switches to the Objects display to display the icon for the new custom object. This icon is titled ªDistributorº.

Control-drag a connection from the Info command in your application's main menu to the Distributor icon. The Inspector panel shows the Connections display for the MenuCell Inspector. Double-click the **showInfoPanel:** action to make the connection. Now, whenever the user chooses the Info command, a **showInfoPanel:** action message will be sent to the Distributor object. The Distributor object will then have to load the auxiliary nib file that contains the Info panel. Save the TextEditor nib file before proceeding.

To build the auxiliary nib file (which is known as a ªmoduleº), choose the New Module command from the Document menu. This command opens a submenu of module types. Choose the New Info Panel command. A new File window opens and a template Info panel appears.

Figure 17-2. Info Panel Template

Customize the text in the panel by changing the application name to ªTextEditorº and by adding your name to the byline. Save the interface you've created in a file named **Info.nib**, in the same directory that holds **TextEditor.nib**. (For example, if your language preference is set to English, this directory will be **~/TextEditor/English.lproj**). Answer Yes to the panel that asks whether you want to add this nib file to the project.

The auxiliary nib file is complete except for connecting the user interface it provides to a Distributor object, the owner of this interface. Before we can connect these two, we must make the Distributor class known within the Info nib file. (So far, the interface to the Distributor class is known only within the TextEditor nib file, where it was declared.)

To make the interface to the Distributor class known within the Info nib file, Interface Builder must parse the class interface file, **Distributor.h**. Switch to the Classes display in the File window for **Info.nib**. Drag to Parse in the pull-down list. In the Open panel that appears, select **Distributor.h** and click OK. The class appears in its proper place in the File window's class hierarchy, and you can view its interface using the Class Inspector.

Now that the Distributor class is known, you can make a Distributor object the owner of the Info nib file. Switch to the Objects display of the File window and select the File's Owner object. The File's Owner Inspector reveals that the file's owner is an instance of the Object class. To reassign the class of the file's owner, click the Distributor entry in the inspector.

Now, connect the Distributor object (the File's Owner object) in the File window to the Info panel. Control-drag a connection from the file's owner to the title bar of the Info panel. In the Connections display of the File's Owner Inspector, double-click the **infoPanel** outlet to establish the connection. Save **Info.nib**.

The graphic part of the interface is done; let's write the **showInfoPanel:** method for the Distributor class. Open **Distributor.m**. (You can do this by double-clicking the class's entry in Interface Builder's class hierarchy browser.) In **Distributor.m**, make the changes that are listed in bold below:

```
#import "Distributor.h"

@implementation Distributor

- showInfoPanel:sender
{
    if (!infoPanel)
        [NXApp loadNibSection:"Info.nib" owner:self];
    [infoPanel makeKeyAndOrderFront:self];
    return self;
}

@end
```

The **showInfoPanel:** method above checks whether an Info panel has already been created. If not, a new one is unarchived from the **Info.nib** file. As the Info panel and the objects are unarchived from the nib file, the Application Kit initializes the Distributor object's **infoPanel** outlet to the **id** of the new Info panel. Finally, this method sends a message to the Info panel (through the **infoPanel** instance variable) to become the key window and order itself to the front of its window tier.

After you save the **Distributor.m** file, the program is ready to compile and test. Click the Run button in Project Builder's project window. When the application begins running, check the operation of the Info command. Notice that the first time you choose the Info command, there's a slight pause before the Info panel appears. However, if you close the panel and choose the Info command a second time, the panel appears instantly. The first time you summon the panel, it must be unarchived from the nib file; thereafter, the panel is simply being ordered on and off the screen list. (If the Info panel doesn't appear when you choose the Info command, quit the program and recheck the connections in Interface Builder.)

So far, this project has demonstrated how to isolate rarely used interface objects in a nib file of their own. The following sections expand on the program to show how to use a separate nib file as a source of document windows for the text editor. Let's take a look at the design of the text editor.

The Text Editor's Design

The text editor has a simple user interface: Through the application's Document menu, a user can open any number of document windows. Text entered in a document window can be cut, copied, and pasted using the Edit menu. With one document window open, the application presents this interface:

Figure 17-3. The Text Editor

The interface you see in Figure 17-3 is created using two nib files. The main nib file contains the specification for the application's main menu and its submenus. An auxiliary nib file contains the specification for a document window and its scrolling text area. The two interfaces are linked by two custom objects (one of the Distributor class and one of the Document class), as shown in Figure 17-4.

Figure 17-4. The Application's Design

When the application starts, the primary interface objects are created from the specification in **TextEditor.nib** and connected to their owner, NXApp. An object of the Distributor class is also created. So far, only the main menu appears on the screen, although the objects that make up the submenus have also been created. When a user clicks the New command in the Document menu, a **createDocument:** action message is sent to the Distributor object. As you can see in the figure, this object in turn creates and initializes a new object of the Document class, a class you'll define in the process of building this application.

The **init** method in the Document class contains these lines:

```
- init
{
    [super init];
    [NXApp loadNibSection:"Document.nib" owner:self];
    return self;
}
```

Each Document object, as it's initialized, is made the owner of a set of objects specified in the **Document.nib** file. Thus, each time the user clicks the New command, a new Document object along with a new window and scrolling text area are created.

The design introduced here is common for applications that replicate pieces of their user interface. The application's core has its own interface. Similarly, each module minimally consists of a custom object and its interface. When a new module is required, an object within the application's core creates the module's custom object, which loads its own interface. In this way, a module can be independent of the application's core objects, storing any pertinent state information in its owner. If the application needs information about a module's state, it can query the module's owner.

In contrast, recall how the Info panel is implemented. With the Info panel, an object within the application's core, the Distributor object, loads the auxiliary nib file. However, the interface module isn't designed to be replicated (in fact, quite the opposite) nor is there any state information that needs to be retained by the module.

Modifying the Application's Interface

Let's implement the design described above by modifying the TextEditor application created so far. First, since the Info nib file is no longer needed, close it by selecting the File window titled ªInfo.nibº and choosing the Close command in Interface Builder's Document menu.

Next, modify the TextEditor nib file by removing the window object. As explained previously, the application's main nib file doesn't include a document windowÐdocument windows are provided by the auxiliary nib file. Remove the window by selecting it (either by clicking it or by selecting its icon in the File window) and then choosing the Cut command. Since most applications have at least one standard window, a panel opens asking if you really want to remove this window. Confirm that you do.

Now, let's add some commands to the main menu. Click the menu button in the Palettes window and drag a Document menu item to your application's main menu. Position this item immediately above Edit. The Document menu that opens displays more commands than you'll need in this project. Cut all but the New and Close commands from the menu.

This completes the visible part of the interface for the application's core. Save the nib file. Next, we'll modify the Distributor class.

Modifying the Distributor Class

The Distributor class must be modified so that new document windows are created whenever a user chooses the New command from the application's Window menu. When a Distributor object creates each new document window module, it temporarily stores the identity of the module's owner. In a more robust application, the Distributor object would keep track of each module's owner so that it could later ªdistributeº messages from the application's core objects to any one of the modules.

To modify the Distributor class, switch to the Classes display of the File window and select the Distributor class. Next, open the Class Inspector, if it isn't open already. Now, add another action message by clicking the Actions button in the Class Inspector and then entering **createDocument:** in the text field. Click Add Action. Now that the new method has been declared in the nib file, you must add it to the class files.

Editing the Class Files

Double-click the **Distributor** entry in the Files window to open both **Distributor.h** and **Distributor.m**. Add the lines that appear in bold in the listings below. An explanation of these additions follows the listings.

Make these changes to the class interface file, **Distributor.h**:

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

@interface Distributor:Object
{
    id infoPanel;
    id newDocument;
}

- showInfoPanel:sender;
- createDocument:sender;
@end
```

Also, make these changes to the class implementation file, **Distributor.m**:

```
#import "Distributor.h"
#import "Document.h"

@implementation Distributor

- showInfoPanel:sender
{
   if (!infoPanel)
        [NXApp loadNibSection:"Info.nib" owner:self];
    [infoPanel makeKeyAndOrderFront:self];
   return self;
}

- createDocument:sender
{
    newDocument = [[Document alloc] init];
    [newDocument show:self];
    return self;
}

@end
```

Each time a Distributor object receives a **createDocument:** message, it creates a new Document object (the owner of the document window module) and stores the object's **id** in its **newDocument** variable. Next, it sends a **show:** message to the new Document object. As you'll see when you define the Document class, this message brings the module's document window to the front of its tier on the screen and makes it the key window.

As suggested earlier, in a more complex application, the Distributor object might keep track of each Document object it creates so that it can send messages to any one of them. For example, it might use an object of the List or HashTable class to record the **id**s of each of the Document objects it creates.

After you've made these changes to the class files, save them and close the Edit windows.

Connecting the Objects

Now, let's connect the New command to the Distributor object. First, notice that the New command is disabledÐits title is in gray. Interface Builder disables menu items from the menu palette that aren't already connected to some target. To enable the New item, select it and switch to the Attributes Inspector. Click the button titled ªDisabledº to remove the check mark. The New command is now displayed in black.

Next, in the File window, switch to the Objects display. Control-drag a connection from the New command in your application's Document menu to the Distributor object in the File window. The Inspector panel shows the Connections display for the MenuCell Inspector. Make sure the **target** outlet and the **createDocument:** action are selected and click Connect. Now, whenever the user chooses New, a **createDocument:** action message will be sent to the Distributor object.

This completes the main nib file; next you'll create the application's document module. Before going on, save your work and, if you like, clean up the workspace by closing the **TextEditor.nib** file. (Choose Close from the Document menu.) You can also close the Inspector panel.

Creating the Module's Interface

A module consists of a window, a scrolling text area, and a custom object that owns this interface. The custom object will be of the Document class, a subclass of Object that you'll define shortly.

Choose the New Module command from the Document menu. From the New Module menu that appears, choose New Empty. This command produces a nib file containing only the most basic components. You can see from the File window that appears that this module consists only of an owner object and a First Responder.

Drag a window from the Palettes window into the workspace and open the Window Inspector. Change the title of the window to ªDocumentº. Now, drag a ScrollView from the Scrolling Views display of the Palettes window into the document window and resize it so that it covers most of the window's area. Save the nib file you've created so far in a file named **Document.nib**. Also, in the attention panel that appears, confirm that this file should be added to the TextEditor project.

The visible portion of the module's interface is complete; the next job is to define the owner object.

Defining the Document Class

In the File window, switch to the Classes display. Create a new subclass of Object by selecting the Object entry and dragging to the Subclass command in the pull-down list. Using the Class Inspector, name this new class the ªDocumentº class.

The next step is to define the outlets and actions of the Document class. Following the same general steps you took with the Distributor class, give the Document class a **myWindow** outlet and a **show:** action method. Create class definition files for the Document (that is, drag to the Unparse button in the File window) and add these files to the project.

Editing the Class Files

As before, edit the class interface file **Document.h** by adding the line that appears in bold:

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

@interface Document:Object
{
    id myWindow;
}

- init;
- show:sender;
@end
```

Also, make these changes to the class implementation file **Document.m**:

```
#import <appkit/appkit.h>
#import "Document.h"

@implementation Document

- init
{
    [super init];
    [NXApp loadNibSection:"Document.nib" owner:self];
    return self;
}

- show:sender
```

```
{
    [myWindow makeKeyAndOrderFront:self];
    return self;
}

@end
```

The **init** method initializes a new Document object and makes it the owner of the module's interface. The **show:** method sends a **makeKeyAndOrderFront:** message to the window in the interface through the Document's **myWindow** outlet.

Connecting the Objects

A Document object owns the user-interface objects that are unarchived from **Document.nib**. Before you can connect the owner to its interface, you must specify that the owner is of the Document class. Switch to the Objects display in the File window and select the File's Owner object. Using the Inspector panel, click ªDocumentº to assign the class of the owner object.

The owner object is connected to the other objects in the application in two ways: through the **show:** action message that it will receive from the Distributor object and through the **myWindow** outlet that will be initialized to the **id** of the window in the module's interface. You've already written the code in **Distributor.m** that sends the **show:** message to a Document object; that connection is complete.

Connect the owner object's **myWindow** outlet by Control-dragging a connection from the owner's icon in the File window to the title bar of the document window. Select **myWindow** in the Inspector panel's Connections display and click Connect. Finally, save the finished nib file.

Now that the pieces are in place and the connections are established, it's time to compile and test the application. Before you do, you may want to clean up the workspace by closing any of Interface Builder's windows you no longer need.

Compiling and Running the Application

Click Run in Project Builder's project window. If the project file needs to be saved, Project Builder asks if you want to save it before proceeding.

When that application begins running, test its operation. Each time you choose the New command, a new window opens directly on top of the old one. If you click in the scrolling text area, a blinking vertical bar appears, marking the insertion point.

Check other features such as text entry and editing, pasting text between windows of this application (and between this and other applications), and window resizing.

Although this completes the text editor project, this application provides a good basis for exploring other features of the Application Kit. Perhaps the easiest improvement would be to add a Font command to the main menu. You could also, for example, implement the Close command or the Document commands that you previously deleted, such as Open and Save. Or you might make it so each new Document window opens in a location offset from the previous one so that old windows aren't obscured by new ones.