Chapter 10

MULTIPLE NIB FILES

CHAPTER 10 MULTIPLE NIB FILES

Goal

To use multiple nib files for implementing applications with multiple, possibly reusable, components.

Prerequisites

Ability to build a basic single nib file application using Interface Builder.

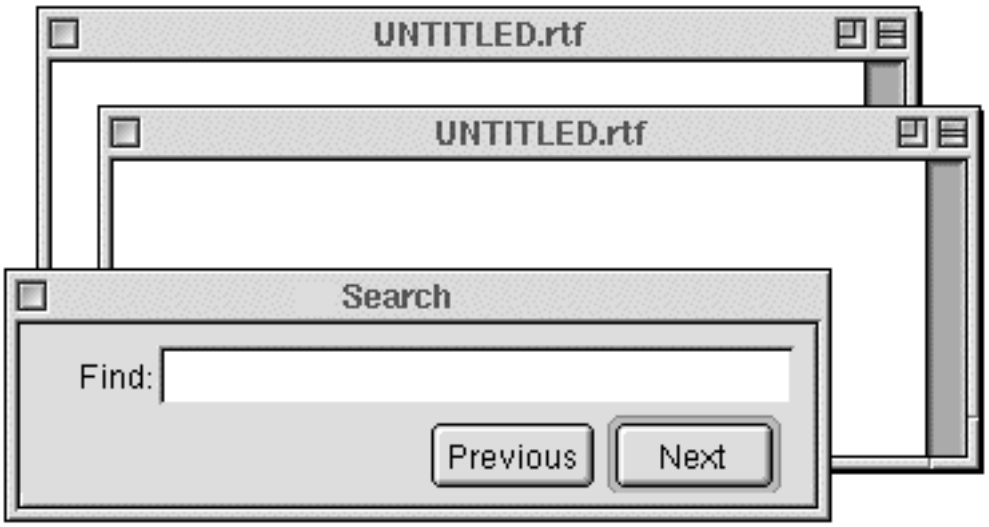
Objectives

At the end of this section, you will be able to:

- Enumerate the advantages of using multiple nib files
- Dynamically load a nib file to instantiate a component
- Explain the role of File's Owner
- Describe the differences between single and multiple instance components with respect to nib loading

Reading

- **Developer Tutorial**
- **Development: Tools and Techniques**
- **NSBundle class reference in the Foundation**

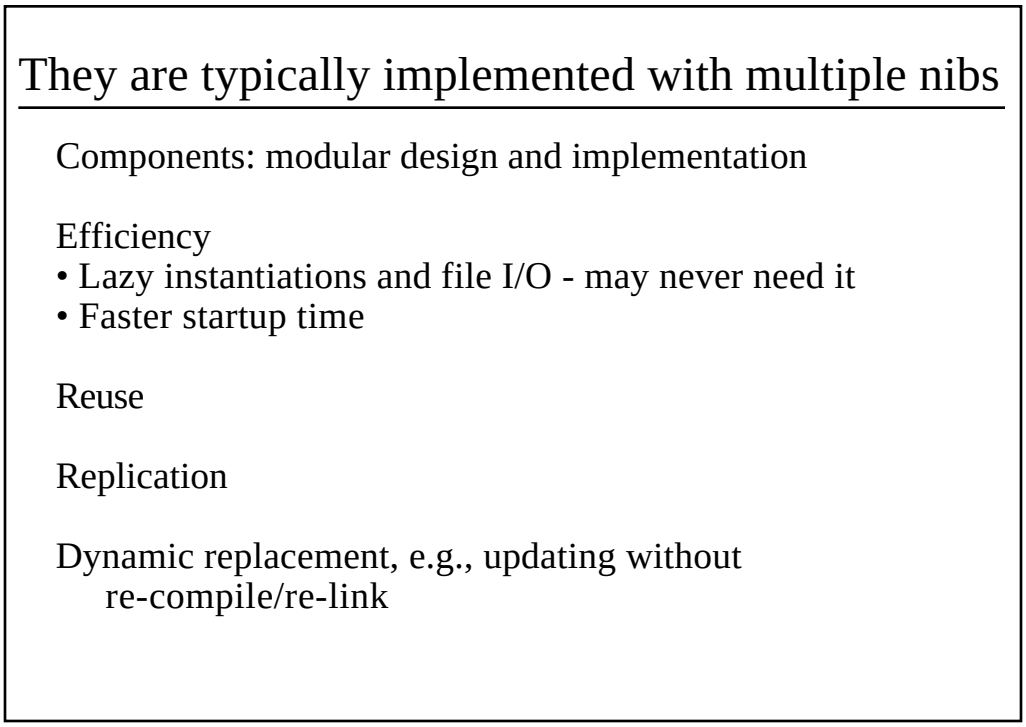


Applications use multiple complex components

A single sophisticated application is likely to incorporate a number of modular components:

- About Panel
- Replicated Document or Window component
- Inspectors
- Preferences
- Find panels and other auxiliary views

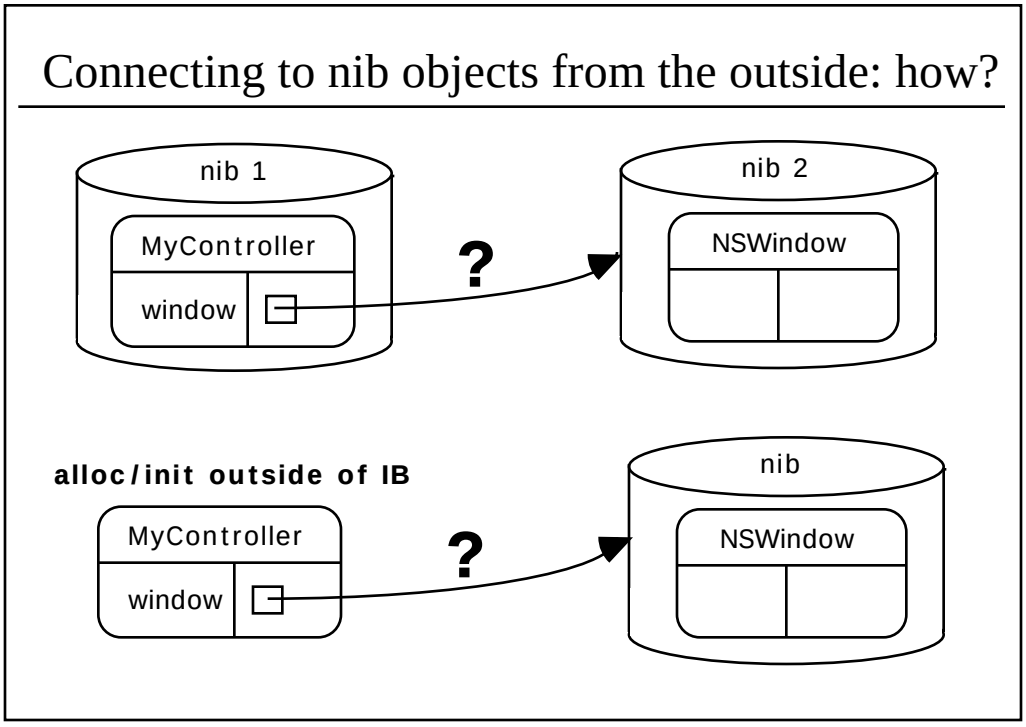
Good software design has traditionally advocated modular design in an effort to encapsulate detail and streamline interfaces between components. Among the many benefits is the possibility that a well designed component can be easily moved from one context to another—it will be reusable. Object-oriented programming aids tremendously in the packaging of such components. A natural way of packaging user interfaces elements into a component is to use a separate nib file.



They are typically implemented with multiple nibs

These are some of the advantages in using a separate nib file for each separate component:

- Modularity
- Efficiency
- Reusability
- Replicability
- Modifiability

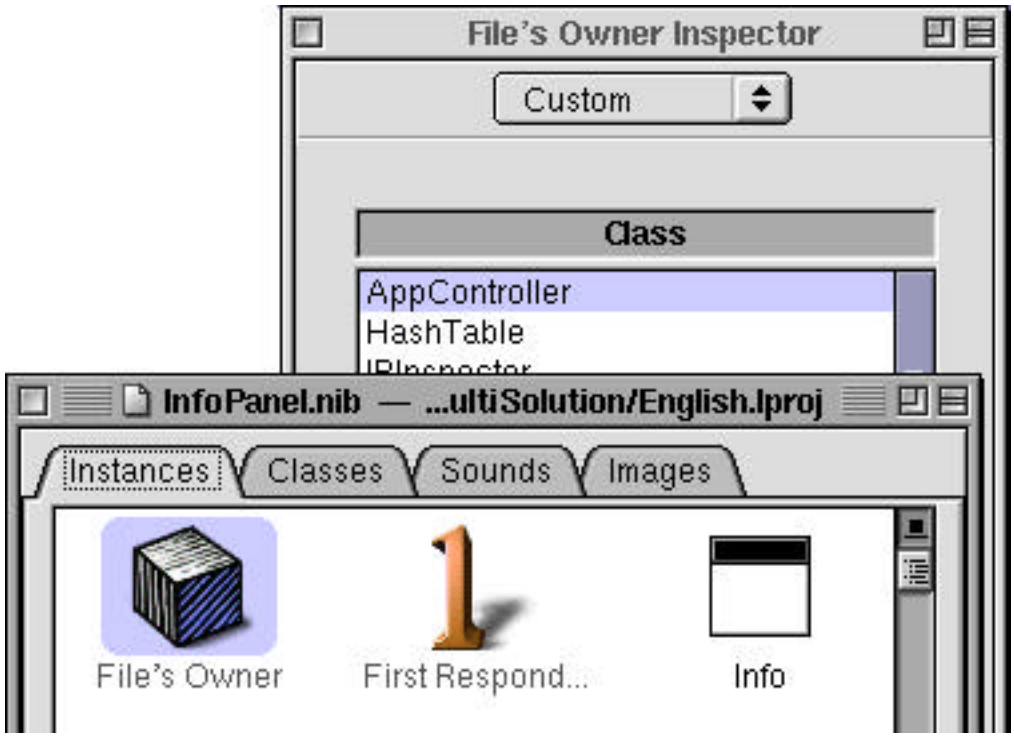


Connecting to nib objects from the outside: how?

Each nib file contains a number of object instances. Interface Builder makes it easy to connect objects within a single nib file. But it is likely, with multiple nib files in a single application, that objects from one nib will need outlets to objects in a second nib file. Since loading a nib means instantiating a set of objects, it is essential that you obtain some handle outside the nib to control these objects once available.

Consider a component that brings up a window announcing the version of the application. A user would activate the component by pressing a menu command. Here is a clear case where a menu item in the main nib needs to message the window or possibly a controller object stored in a second nib.

How is it possible to make these connections?

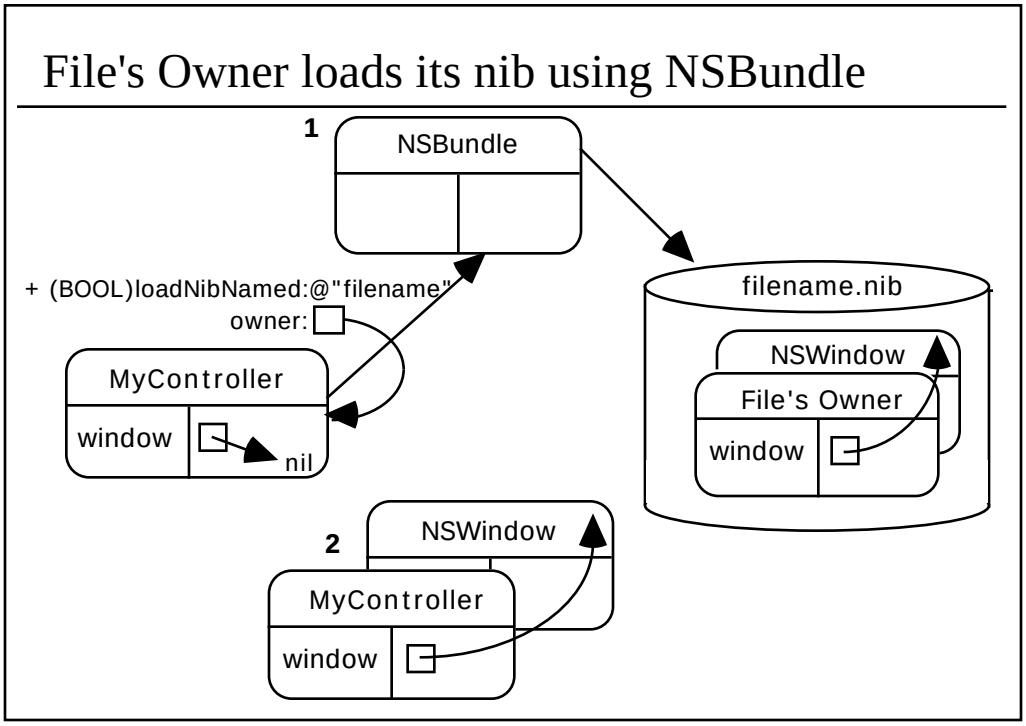


Every nib file has a File's Owner

For every nib file, Interface Builder provides a placeholder (a proxy) that represents exactly one object instance that lives outside of the nib file. It is called the File's Owner. By definition, it must be instantiated by some other means than loading this particular nib. Indeed, it must exist before this nib is ever loaded. It is the "owner" of this nib file in that it serves as the sole bridge the life outside of the nib, or alternatively, a way for the outside world to get inside the nib.

With the inspector, you can configure the class of object that File's Owner will be. It will usually be a custom class that you have created and read into Interface Builder. Since Interface Builder knows the classes interface—its methods and instance variables—it is knowledgeable about the of connections you can make between the File's Owner and other objects within the nib.

If the File's Owner represents an instance of MyController, you can now connect its **window** outlet to the NSWindow instance within the nib. When it's time to load the nib, this connection will be established. Objects within one nib can be automatically connected to exactly one object outside the nib, the File's Owner.



File's Owner loads its nib using NSBundle

The File's Owner proxy in every nib file is a promise: when it's time to load the nib file, an object instance will be provided that:

- Is an instance of the same class as that shown in the inspector for File's Owner
- Needs to be connected to objects within the nib

The NSBundle class loads a nib file with the **loadNibNamed:owner:** method. It takes the real File's Owner object instance as an argument. Once the nib has been loaded and the method returns, File's Owner outlets connected to objects in the nib are now properly set and ready to go. Any objects in the nib pointing back at File's Owner now point to the real object instance, in this case MyController.

What about **awakeFromNib**? As usual, all objects inside the nib that responded were sent **awakeFromNib**. MyController, though intimately associated with the nib, was not archived within the nib. While it does receive **awakeFromNib** as File's Owner, it is not typically used here. Upon returning from **loadNibNamed:owner:**, you are assured that all nib objects have been instantiated and connected. There is no need for an explicit message. Any nib finalizing can be performed by MyController at this point.

Code example: loading a nib file

```
- (void)show:(id)sender
{
    NSString *file = @"MyNib";

   if (!window) {
      if (![NSBundle loadNibNamed:file owner:self] {
         NSLog (@"Unable to load nib '%@' ", file);
         return;
      }
   }
   [window makeKeyAndOrderFront:nil];
}
```

Code example: loading a nib file

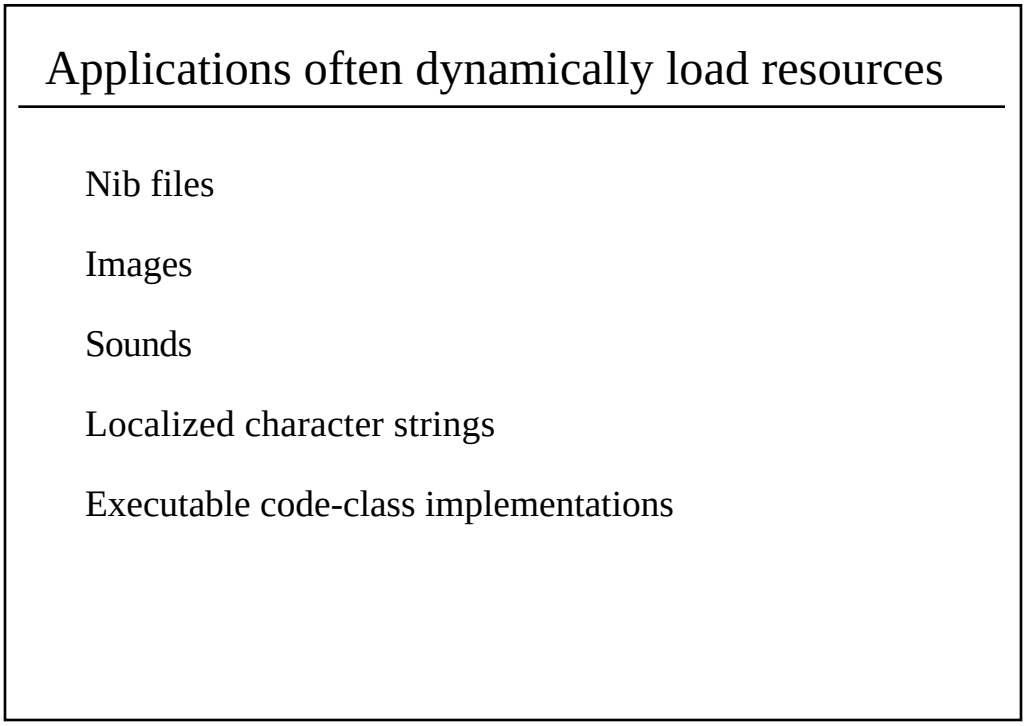
As soon as a client wants your component to show itself, it's time to load the appropriate nib file.

The component controller implements the **show:** method with two assumptions:

- This controller is the nib File's Owner
- It has a **window** outlet that is connected to the NSWindow instance inside the nib once the nib has been loaded

Until the nib is loaded, the NSWindow instantiated and the outlets connected, the controller's **window** outlet is nil. The outlet is used as a flag to determine whether or not the nib has been loaded. If it is nil, load the nib. If it is non-nil, the nib has already been loaded and the window instance is available. It would be incorrect to load the nib multiple times from the same controller instance.

If we fail to load the nib successfully, we do the courtesy of logging an error with **NSLog()** and return. Whether we just loaded the nib or did so several calls to **show:** earlier, we must now make the window visible and active. It may be that the window was simply ordered off screen as the result of a close. In this case, **show:** orders it back on screen.

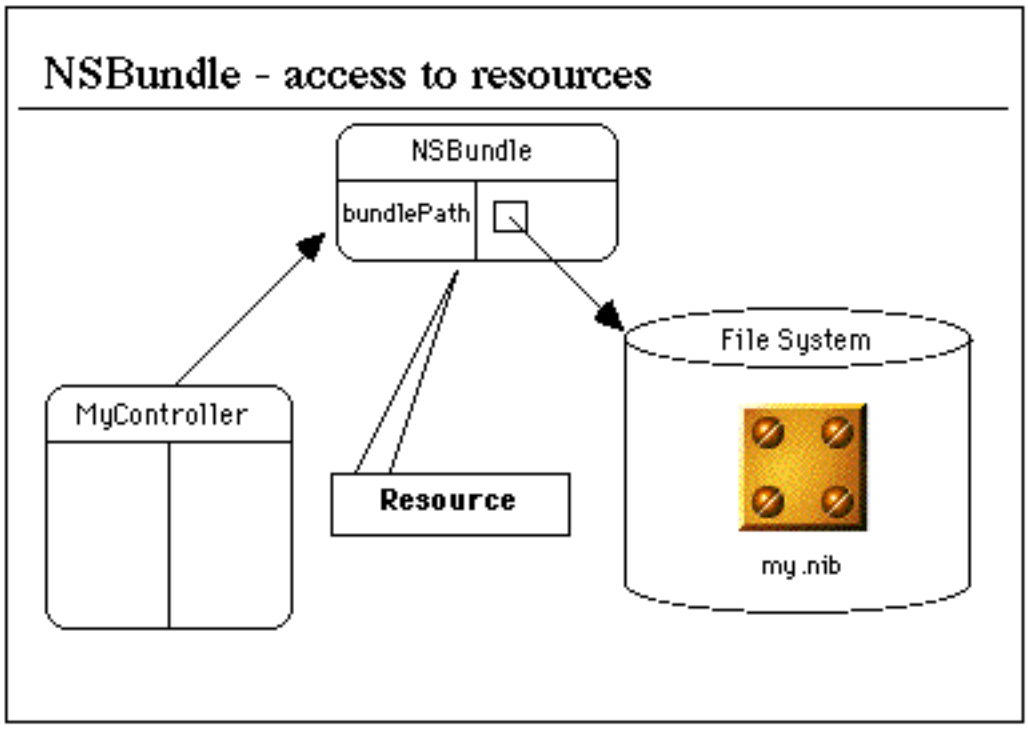


Applications often dynamically load resources

Most applications contain a large number of resources that are only needed under certain circumstances. For efficiency and maximum flexibility, an application can dynamically load a wide variety of resources:

- Nib files—modular user interface components used by your application.
- Images—applications often use a fixed set of images in the user interface but perhaps only conditionally. Apps don't need them until it is necessary.
- Sounds—same idea.
- Character strings—applications localized to multiple languages need only speak one at a time.
- Code—you can even dynamically load class implementations.

All of the reasons for using multiple nib files apply to each of these resources as well. Until they are loaded, your application uses less memory. If a resource is dynamically loaded, it can just as easily be dynamically replaced, modified, updated or extended. It is even possible to allow some measure of open-ended extensibility to an application. In the future, you may dynamically load and interact with a component that you never planned on, nor even care too much what it does or how it does it!



NSBundle—access to resources

You can think of an application not so much as a fixed, static thing but as a loose collection of resources and components stored together somewhere in the file system. An application built in this way is really a bundle of resources available for dynamic loading.

NSBundle is a Foundation object that represents a file system location that bundles these resources together. You can ask an NSBundle instance for the pathname to a resource or you can simply ask that it load the resource now, incorporating it into the world of your active application.

While your application is itself a bundle, you are not restricted to it alone. It is possible to have multiple bundles either contained within your application bundle or located elsewhere in the filesystem. It is possible to imagine a bundle of resources shared by many applications. It might be useful to have one bundle of quickly changing resources dynamically accessed by a separate application bundle that changes infrequently or not at all. Bundles, with their ability to dynamically load resources, support the implementation highly dynamic applications.

NSBundle methods

```
+ (NSBundle *)mainBundle;

+ (NSBundle *)bundleWithPath:(NSString *)path;

+ (BOOL)loadNibNamed:(NSString *)name owner:(id)owner;

- (NSString *)pathForResource:(NSString *)name
             ofType:(NSString *)ext;

- (NSString *)localizedStringForKey:(NSString *)key
             value:(NSString *)value
             table:(NSString *)tableName;

- (Class)classNamed:(NSString *)className;
```

NSBundle methods

These NSBundle methods address the full list of potentially loadable resources. Be careful to note that some are factory methods while others instance methods.

Every application is itself a bundle. It is where your application's executable image lives as well as the main nib loaded automatically at launch time. You can obtain a bundle instance bound to this path with **mainBundle**. Alternatively, you can instantiate a bundle with an arbitrary file system path using **bundleWithPath:**.

You load a nib file using a factory method and no specification of a path, i.e. without explicit reference to a bundle instance. Which bundle is used? It uses the bundle that supplied the class implementation for the File's Owner object which is passed as the second argument. This might be a separate bundle you have created or more commonly the main bundle itself.

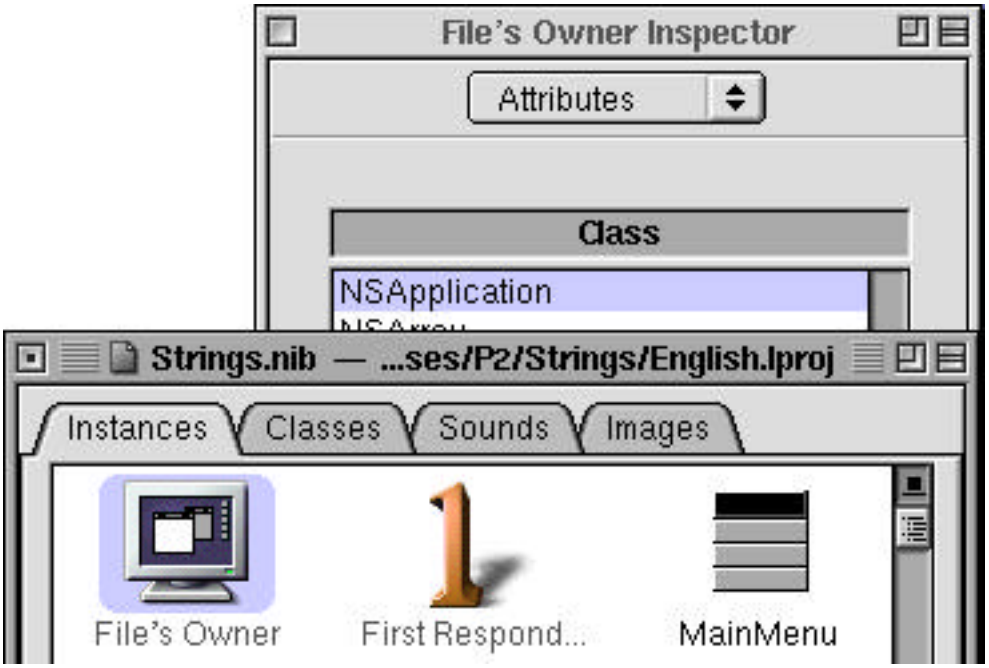
While a bundle can automatically load a nib file, a localized string, or even a chunk of class implementation code, it can't know everything you do with your resources. Often, you simply need a full pathname for a resource. What you do with it is your business. NSBundle will locate named files of a specified type—in other words, with the specified file extension—returning its full pathname to you. This automatically includes locating one of several different versions of the file, each for a specific language in the case of a localized application.

Using NSBundle, it is possible to dynamically load not only the nib file but the class implementation, the compiled code, of the File's Owner controller as well. A complete component, the controller and its nib, can be a loosely coupled application resource packaged separately in a bundle. The bundle can even be replaced independently of the application itself. Through dynamic loading, changes, and updates will be incorporated into the application the next time it runs. Here is a sample method for dynamically loading an entire component:

```
- (void)showComponent:(id)sender
{
   id component; // likely to be an ivar, shown here for clarity
   Class componentClass;

   // find the bundle in the mainBundle
   NSBundle *bundle =
      [NSBundle bundleWithPath:[[NSBundle mainBundle]
      pathForResource:@"Component" ofType:@"bundle"]];

   // alloc/init and show:
   if (componentClass = [bundle principalClass]) {
      component = [[bundleClass alloc] init];
      [component show: nil];
   }
}
```

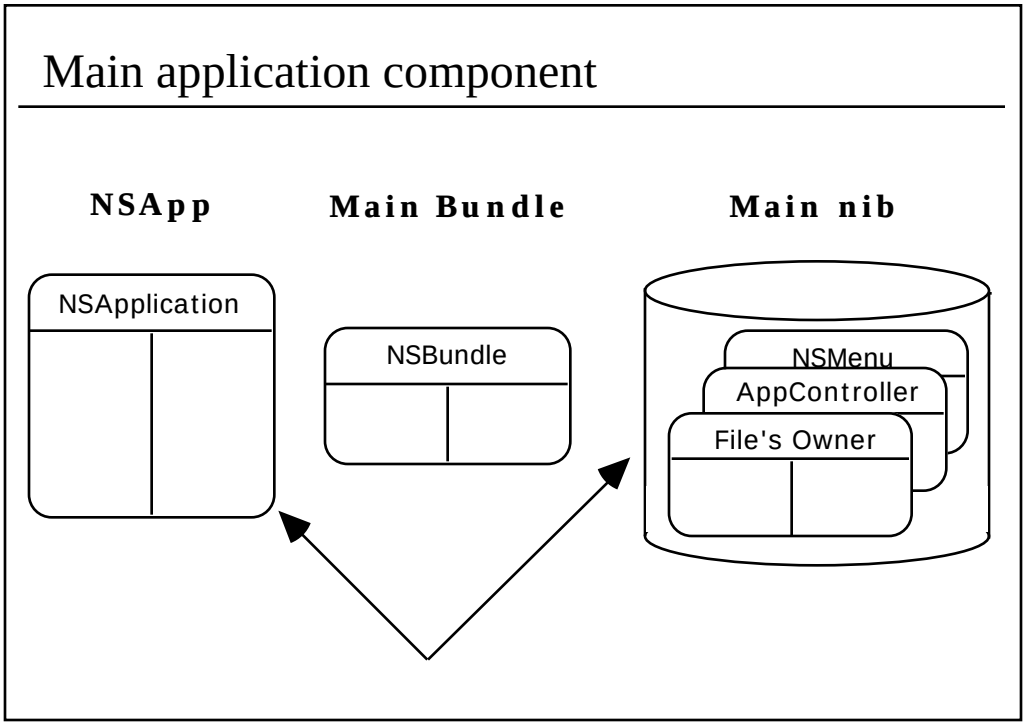


The main nib: who loads it, who owns it?

The main nib file of your application is transparently loaded by your project's **main()** function which is generated by Project Builder—it calls the Application Kit **NSApplicationMain()** function. The name of this nib file is listed under "Project Attributes" on the Project inspector and is configurable. This is all automatically established for you when you first create a project.

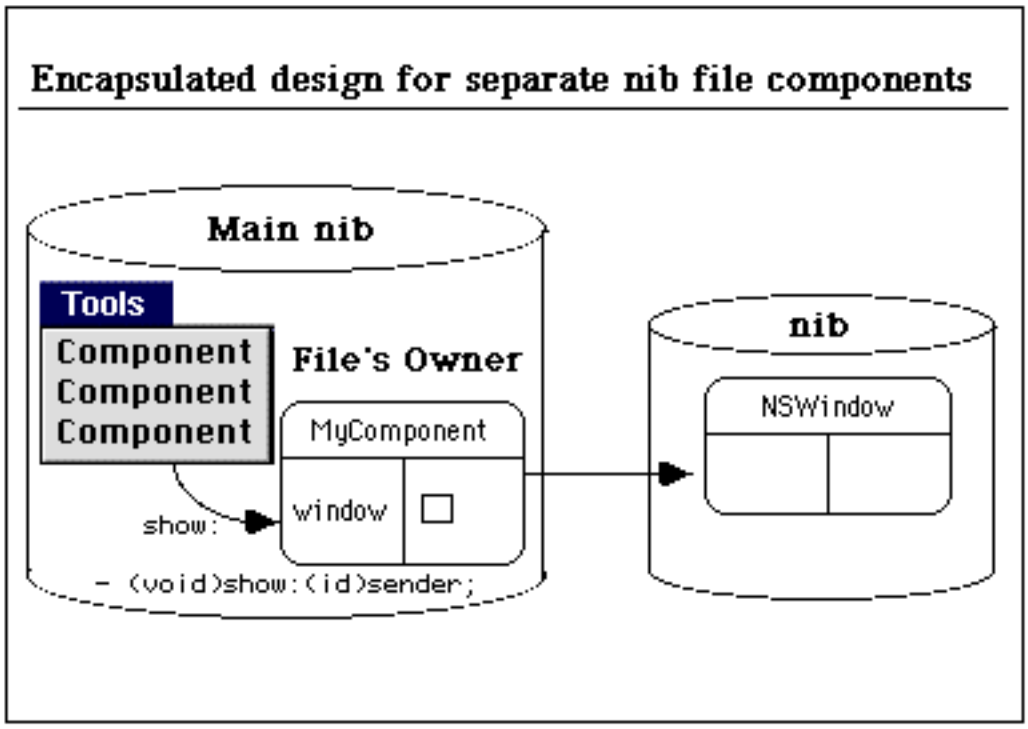
What object does the File's Owner icon represent in the main nib file? Every application is born with one and only one instance of NSApplication. The Interface Builder inspector confirms File's Owner's type to be just that. When you discover the reasons for doing so, you can easily make connections between the NSApplication instance and the objects within the main nib. The most common example is connecting your application controller as the **delegate** of NSApplication.

You should generally never reassign the File's Owner of your main nib.



Main application component

Each application has a main component comprised of the global NSApplication instance, the main nib and the main bundle enclosing them. The main nib includes the File's Owner proxy for NSApplication, the main menu instance and any number of additional custom object instances. Your application controller customizes the behavior of the main component by providing menu item target/action, and serving as a delegate to NSApplication. It is useful to view this as a modular component in itself, usually only one of several that comprise a sophisticated application.



Encapsulated design for separate nib file components

Here is a simple design pattern that applies well to most separate nib file components such as find panels, inspectors, preferences, and the like. The design incorporates two pieces:

» The Controller—an object outside the nib that serves as the File's Owner

» The user interface—the nib file

When is the nib file actually loaded? When the controller needs it to be there. Other application objects request the services of the component via the controller. Typically, this might be as simple as sending the **show:** message. Until the user actually needs to use the interface, there is no need to load the nib and instantiate the objects it contains.

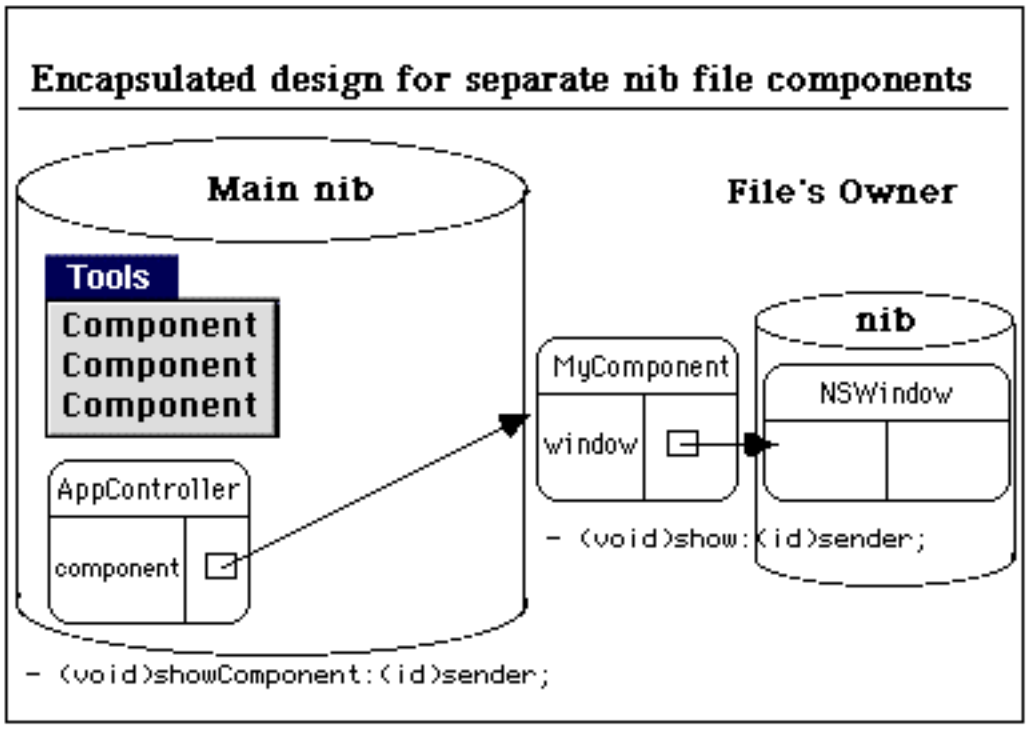
Where and how does the controller itself get instantiated? Remember, it is always outside the component nib. It may be instantiated in another nib, commonly the main nib. Or, it might be dynamically instantiated with **alloc** and **init** in which case it does not live within any nib file at all.

If it is instantiated in the main nib file, there are some implications to consider:

» The target/action menu item can connect directly to the controller

» Nib loading should happen in **show:**, not **init**

» The controller implementation cannot be dynamically loaded (e.g. via NSBundle)



Encapsulated design for separate nib file components

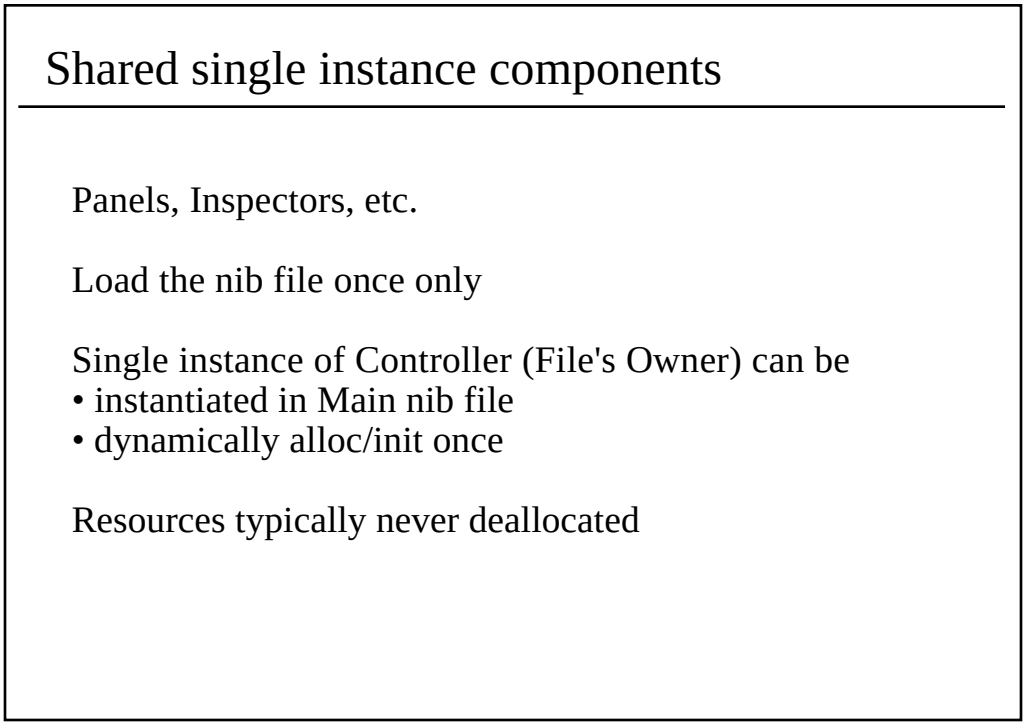
This second design is used for replicated components such as document windows. It is also well suited to single instance components and provides more flexibility than the previous arrangement. Menu actions or component requests go through the application controller which in turn passes them to the component controller through an outlet. The controller is not instantiated in any nib, but is dynamically allocated once, the first time the application controller needs it. The first time, AppController's **component** outlet is nil:

```
- (void)showComponent:(id)sender
{
   if (!component)
      component = [[MyComponent alloc] init];
   [component show: nil];
}
```

For replicated components—there may be multiple instances of the controller—AppController would need a collection outlet such as NSArray to store the multiple outlets.

Even for single instance components this design has advantages:

- The component controller is not even allocated until it is needed.
- The controller implementation—the compiled class code—can now be dynamically loaded (via NSBundle) as well as the nib it owns.

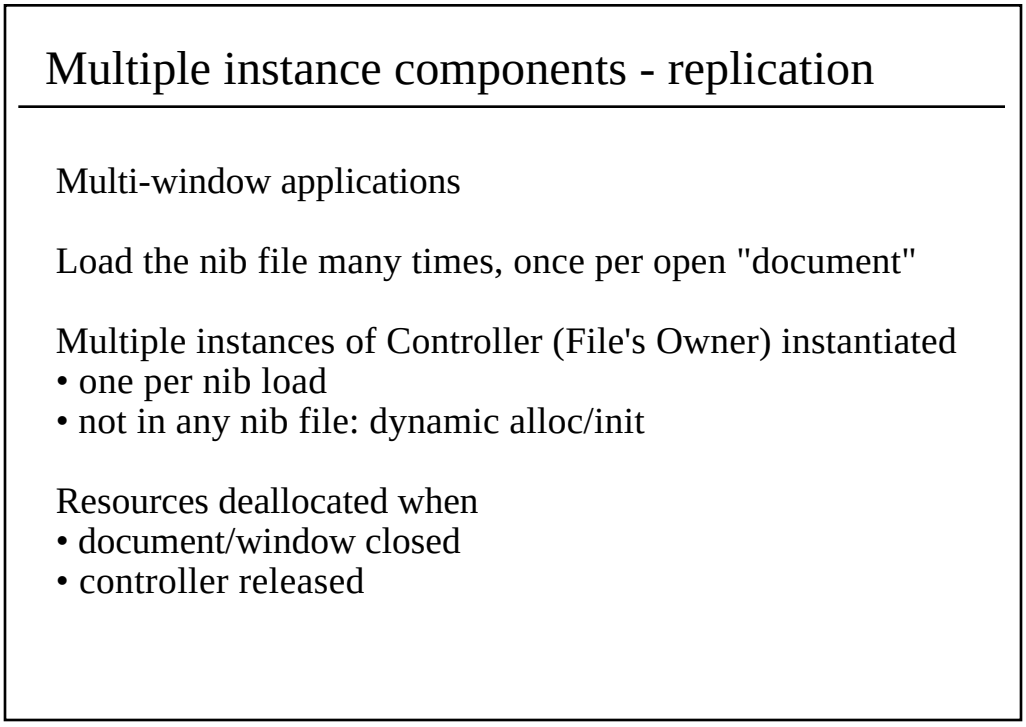


Shared single instance components

Most components are single instances shared throughout the appliction: you have one main menu, one preferences panel, one find panel. Since every time you load a nib file it instantiates everything in the nib, you only need do so once in the entire life of the application. This is based on the assumption that you will never dynamically free such a component once it has been created.

It is likely in this case that the component controller, the File's Owner outside of the nib, will be instantiated within the main nib. This allows you to make connections between menu items and the controller, or give your application controller an outlet to it. For completely lazy instantiation and the possibility of bundle-loading the controller implementation itself, File's Owner is not instantiated in the main nib but dynamically allocated once.

The component nib is not loaded, however, until someone really needs it. The controller is present, ready to go. But it needn't actually load the nib until someone asks it to **show:**.



Multiple instance components—replication

Consider your favorite editor. It probably permits you to have several different documents open at the same time. Each one appears in its own window. If there are controls on that window, they are probably duplicated faithfully on each open window you have. While the contents your are editing is unique within each window—the file you are editing—the user interface components are the same, cloned for each open file.

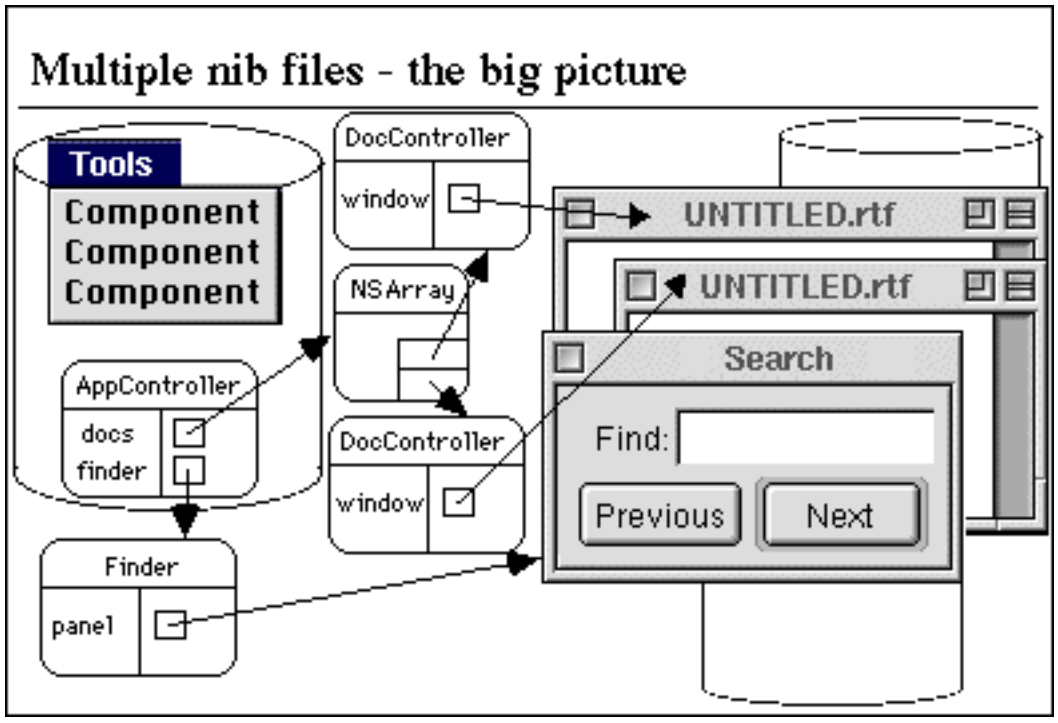
Once you have implemented a nib file and controller pair, it is easy to replicate several instances of the component:

» Instantiate a new controller via **alloc** and **init**

» Message the controller, asking that it **show:** itself

Since new controllers are dynamically instantiated at run time, no single instance is likely to live statically within in the main nib. And unlike a single instance component that lives for the life of the application, this kind of component is most likely going to be freed when its no longer useful. This might happen when you are done editing the file and you close its window.

You can configure a window to free itself and all the user interface objects it contains when the user closes the window. Your component controller will find out about the close and can take care of freeing itself. Some nib objects need special consideration and will be addressed later.



Multiple nib files—the big picture

Let's return to the picture at the beginning of the chapter but this time include some object diagrams and suggest how it might be divided into multiple nib files. There are three different nib files in the picture:

- Main nib—contains NSMenu and AppController instances. NSApplication (not shown) is the File's Owner.
- Find Panel nib—contains the Find Panel instance. File's Owner is the Finder object. It may be instantiated in the main nib or, as implied here, dynamically allocated (once, lazily) the first time it is needed.
- Document nib—loaded multiple times. It contains the document Window instance. File's Owner is a dynamically allocated DocController instance.

This suggests a general design template applicable to many different applications assembled from separate nib-based components.

Important ideas from this section

» Applications are typically assembled from multiple components. These are best implemented as separate nib files.

» File's Owner is the single object instantiated outside of a nib file. It typically loads the nib file and serves as a bridge for connections to objects within the nib file.

» NSBundle provides API for dynamically loading application resources, including nib files.

» The nib file for single instance components, such as an About Panel, is loaded once. Its File's Owner is usually instantiated in the main nib.

» The nib file for multiple instance components, such as a DocController in a multi-window application, is loaded many times. Its File's Owner object is dynamically created each time a new component instance is needed. It loads its own nib.

» Every application has a main nib which is automatically loaded at application launch time. The single instance of NSApplication is its File's Owner.

Class featured in this section

NSBundle

REVIEW MULTIPLE NIB FILES

1. Why would you want to have an application with multiple nib files?

2. How can you connect an object in one nib file to an object in a different nib file?

3. What's the role of the File's Owner in a nib file?

4. Describe the relationship between a nib file and a bundle.

5. In the case of a single instance component like a Find panel, how does the File's Owner know if the nib file has already been loaded?

6. How can File's Owner tell if the nib file failed to load? What are some reasons why this might happen? What should File's Owner do about it?

EXERCISE 10.1 MULTIPLE NIB FILES—ADDING AN ABOUT PANEL

Virtually every application has a facility which provides some information about the application, such as title, version number and author. This is commonly called an About or Info panel. In this exercise you extend the application from the previous exercise to include an About panel using a second nib file. The panel is accessible from the application's main menu.

Objectives

After completing this exercise, you will be able to:

- Build a separate panel with controller component using Interface Builder
- Understand and use File's Owner in Interface Builder
- Connect menu actions directly to other controller objects

Exercise

1. Make a copy of the previous project to preserve the original. Open the new project and change the project name—with the Project Attributes Inspector—to **AboutPanel**, to distinguish it from earlier versions. Save it.

2. Create a new Component subproject called **AboutPanel**. A subproject contains its own separate suitcases for classes, header, interfaces and so on. It organizes your project into separate modules. When you build the project, each subproject is built and incorporated into the final application.

3. From Interface Builder, create a new nib file of type "About Panel" using the New Module menu item on the File menu. Customize the appearance of the panel as you like. Save the nib file in the folder for the AboutPanel subproject. When prompted, insert it into the project.

4. This nib will need its own custom controller object. Subclass NSObject and call the class **AboutPanel**. Include an outlet for the panel. You also need an action to display the panel upon request, so add a method called **show:**.

5. Using the Attributes inspector, make the File's Owner class your AboutPanel class. You can then make a connection from the File's Owner to your panel. Note: do NOT instantiate a controller object in the AboutPanel nib itself.

6. From the classes window in Interface Builder, create the files for your controller object and add them to the project. Save the interface in the AboutPanel component, and switch back to Project Builder.

7. Implement the **show:** method in **AboutPanel.m**. It should check the **panel** outlet and if nil, load the nib file with self as owner. This makes the File's Owner object the same class as it was defined in the AboutPanel nib file and will connect the **panel** outlet to the NSPanel instance. In all cases, **show:** should display the panel using **makeKeyAndOrderFront:**.

8. Configure a menu item to display the panel. Your application already has an About menu item—under the Apple menu—so it's only necessary to make the target/action connection.

    - Read **AboutPanel.h** into your main nib file by dragging it into the Interface Builder instances window for **Strings.nib**, and instantiate an AboutPanel.
    - Connect the About menu item to the AboutPanel instance's **show:** action.

9. Make sure everything is saved and then build the project. Check that the About panel appears when you choose the About menu item. Close and redisplay the panel to verify that it works. Press the About menu item more than once, even after the panel is visible. Make sure you do not see multiple panels.

Enhancements

Try an alternative design for the About Panel. This design forwards main menu messages to the AboutPanel controller via the main AppController, rather than making the connection directly.

- Add an **aboutPanel** outlet to AppController.
- Add a **showAboutPanel:** action to AppController.
- Remove the AboutPanel instance from the main nib and connect the About menu item to the AppController's **showAboutPanel:** action.
- Using Project Builder's file attributes inspector, make the **AboutPanel.h** header file a Project Header file. This means the header file is available project-wide—it can be imported by other files in the project even if they are in different subprojects. You can now add an import statement for the class to your **AppController.m** file.

» Now write AppController's **showAboutPanel:** method. If the **aboutPanel** outlet is nil, create an instance of your AboutPanel class and then send it the **show:** message.

```
- (void)showAboutPanel:(id)sender
{
   if (!aboutPanel)
   {
      // first time through alloc/init AboutPanel instance
      aboutPanel = [[AboutPanel alloc] init];
   }
   [aboutPanel show: nil];
}
```

» Save all the modified files and build the project. Test the application—check that the info panel still appears.

Once you have finished the previous enhancement, it is reasonably straightforward to make the AboutPanel a loadable bundle subproject. This design makes the whole Info panel component lazy, including the loading of the code and the nib, which will keep the application as small as possible. You will have to create a new Bundle subproject and copy all the files into it.Then delete the old AboutPanel subproject. Use the following method implementation in AppController:

```
- (void)showAboutPanel:(id)sender
{
   if (!aboutPanel) {
      // first time through
      Class class;

      // find the bundle in the mainBundle
      NSBundle *bundle =
         [NSBundle bundleWithPath:[[NSBundle mainBundle]
         pathForResource:@"AboutPanel" ofType:@"bundle"]];

      // alloc/init AboutPanel instance
      if (class = [bundle principalClass]) {
         aboutPanel = [[class alloc] init];
      }
   }
   // show it
   [aboutPanel show: nil];
}
```