Interface Builder Plug-In Programming Guide

Tools > Interface Builder

2007-07-18

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Interface Builder Plug-in Programming Guide

Although Mac OS X provides many useful views and controls for creating user interfaces, there may be times when you need to extend the basic set to achieve an appearance or behavior that is more appropriate for your application. The problem with custom views is that you generally cannot see how they will look in your user interface until that part of your application is running. Even when your application is running, making changes to the attributes of custom views requires modifying your code and rebuilding your application before you can see those changes. For large applications, turning around such changes can be slow and frustrating. Luckily, Interface Builder provides a way for you to integrate custom controls into the Interface Builder environment, build those controls into your application's user interface, and see the results immediately.

The infrastructure for integrating custom controls in Interface Builder version 3.0 has improved significantly over previous versions of the application. The current infrastructure makes it possible to integrate new controls quickly and add support for more advanced features, such as inspectors, in stages. The process for creating inspectors has also improved dramatically and lets you focus on the new attributes exposed by your custom objects.

You integrate controls through the use of plug-ins. Each plug-in provides Interface Builder with design-time information about one or more custom objects, including where to find them and how to change their attributes. Upon loading your plug-in, designers can then drag your custom controls from the Interface Builder library window and use them to build their interfaces. Your controls can also be saved in the resulting nib file and instantiated at runtime.

Interface Builder supports the creation of plug-ins for Cocoa-based views, controls, and objects only. You currently cannot create plug-ins for Carbon-based objects.

Organization of This Document

This document guides you through the process of creating plug-ins for use with Interface Builder version 3.0. The chapters in this book are intended to be read more or less in order. Early chapters provide basic information that all plug-in developers need to know, while later chapters provide more advanced topics.

- "Anatomy of a Plug-In" (page 9) provides an overview of Interface Builder plug-ins, discussing their structure and how they interact with other code modules.
- "Plug-in Quick Start" (page 17) provides a quick tutorial of how to build a basic plug-in.
- "Preparing Your Custom Objects" (page 25) describes the steps you need to take to prepare your custom controls, views, and objects for incorporation into the Interface Builder environment.
- "The Plug-in Object" (page 31) describes the purpose of the plug-in object and how you tailor it for use with your custom objects.
- "Inspector Objects" (page 39) describes the process for creating inspectors, which are used to modify the attributes of your objects in the Interface Builder application.
- "Advanced Techniques" (page 47) describes additional tasks associated with integrating more complex objects and views into Interface Builder.

Note: This document does not provide information on how to create plug-ins for Interface Builder 2.5 and earlier.

Anatomy of a Plug-In

This chapter provides a high-level overview of the plug-in model used by Interface Builder and discusses some of the terminology associated with building plug-ins. If you are interested in integrating any custom objects into the Interface Builder environment, you should read this chapter to get a basic understanding of how Interface Builder plug-ins work.

Note: You can use Interface Builder plug-ins to incorporate custom Cocoa-based objects only. You cannot integrate custom Carbon-based objects.

Interface Builder and Plug-ins

Interface Builder is a tool for building application user interfaces visually from a standard set of user interface components, including windows, menus, views, controls, formatters, and controller objects. The Interface Builder application comes configured with the standard controls available to all Cocoa and Carbon applications. Although the standard controls are useful for many applications, they may not be sufficient in all cases. You might want to create new controls or customize the appearance of the standard system views and controls. Instead of creating new controls, you might want to speed your design process creating customized configurations of the standard controls. For all of these goals, you can use Interface Builder plug-ins.

You install plug-ins from Interface Builder's preferences window. Once installed, the plug-in acts as a liaison between Interface Builder and the code for your custom views and objects. The plug-in specifies the initial configuration of your views and objects and provides the means to manipulate the attributes of those views and objects at runtime. The plug-in is therefore responsible for the following basic jobs:

- It identifies which of your custom objects to add to the Interface Builder library window.
- It tags various attributes of your custom objects as being user manipulable. (Interface Builder uses this information to implement several housekeeping tasks related to your objects.)
- It provides the user interface required to manipulate the attributes of your objects.

Deciding When to Create a Plug-In

Before creating an Interface Builder plug-in for your own custom objects, you should think about whether a plug-in is an effective use of time for you. In particular, carefully consider the following:

- Are your custom objects going to be used by only one application?
- Do your custom objects rely on state information found only in your application?
- Would it be problematic to encapsulate your custom views in a standalone library or framework?

If you answered yes to any of the preceding questions, your objects may not be good candidates for a plug-in. The main purpose of integrating custom views and controls into Interface Builder is to streamline the process of creating and customizing your user interfaces. However, putting your views and controls in a plug-in takes effort too. If you plan to use a view for only one application, it might not be worth the extra effort needed to create a plug-in for it. Similarly, if your views are too tightly entwined in your application logic, extracting them from that logic may require more effort than is worthwhile. Custom objects must be able to operate outside of your application environment so that they can be integrated into Interface Builder.

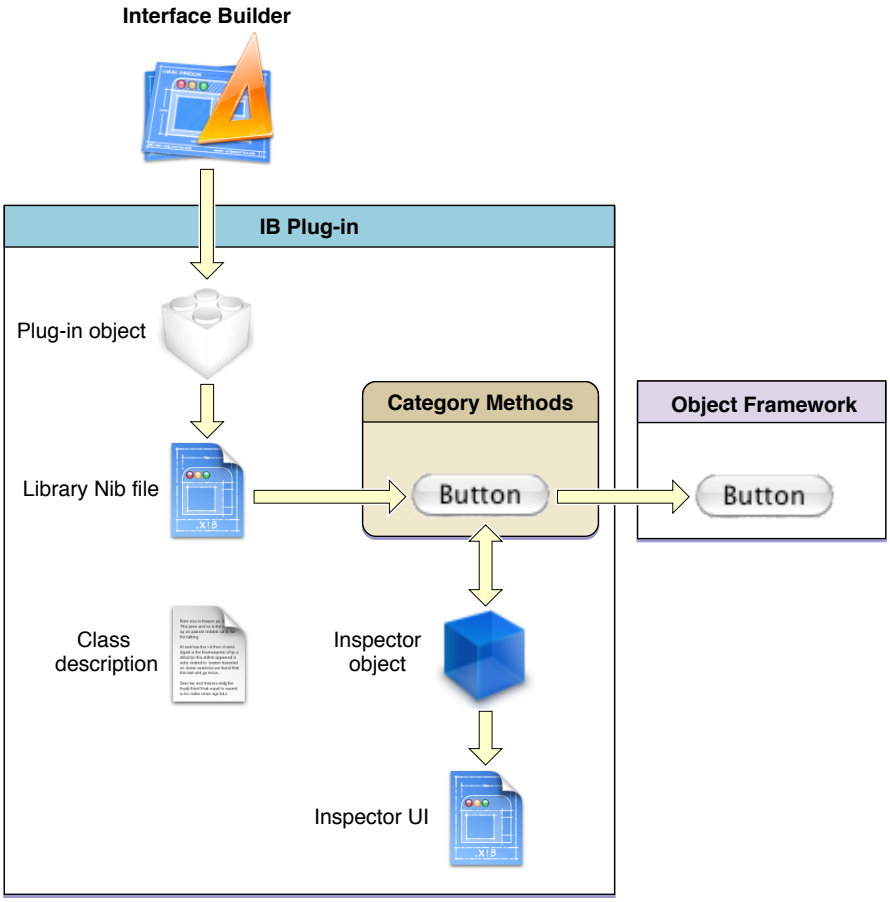
The views and controls that make good candidates for inclusion in a plug-in are those that can stand on their own and be used by multiple applications. Each view or control you design should be self-contained and not make any assumptions about the state of its host application. Whenever possible, views should also avoid making assumptions about the existence or state of other views, although in some cases knowing about other views may be necessary. For example, a scroll view is typically grouped with other views, including a clip view and scrollers. If you do have views whose behavior is tightly intertwined, you may need to deliver them as a preconfigured group rather than as separate pieces.

The Structure of a Plug-in

An Interface Builder plug-in is a bundle that contains a loadable executable file and some supporting resources. Nearly all Interface Builder plug-ins actually contain at least one nib file and many can contain several nib files. (Interface Builder relies on nib files whenever possible to simplify the plug-in creation process.) The bundle directory for an Interface Builder plug-in must have the `.ibplugin` extension.

Figure 1-1 shows the high-level structure of an Interface Builder plug-in and some of the other code modules with which it interacts. A plug-in bundle links against the object framework that contains the code for the objects being added to Interface Builder. Inside the plug-in itself are the handful of objects and files (including the plug-in object, inspector objects, nib files, and class descriptions) that bridge the gap between Interface Builder and your custom framework. Any category methods that are related to Interface Builder but defined on your custom objects should similarly be included as part of your plug-in and not as part of your object framework.

Figure 1-1 Objects associated with an Interface Builder plug-in



The pathways through the preceding figure show the routes taken by Interface Builder to discover objects inside of your plug-in. Once acquired, Interface Builder may cache references to various objects for easier access later. For inspector objects, the route is shown as bidirectional to reflect the interactive nature of those objects with the current selection.

From the figure, you can see that the discovery of all custom objects occurs through the plug-in object and its associated library nib files. Interface Builder integrates the contents of each library nib file into the library window. As items are dragged out of the library window and into a document, Interface Builder uses the category methods of the dragged object to locate other needed objects. For example, when an instance of your object is selected in a document, Interface Builder checks the category methods to see if an associated inspector is available, and if so, assembles the pieces of the inspector user interface needed to represent your control.

Key Plug-in Objects and Files

The following sections describe some of the key objects from Figure 1-1 (page 11) that you are responsible for creating. For more information about the classes used to create these objects, see *Interface Builder Kit Framework Reference*.

The Plug-in Object

The plug-in object is the main entry point to your plug-in. Interface Builder uses this object to obtain your plug-in's name and the list of custom objects it to integrate into the library window. This object also manages some general plug-in features, such as your plug-in's preferences. Every Interface Builder plug-in must have a plug-in object, which is a subclass of the `IBPlugin` class.

At a minimum, every `IBPlugin` object must implement the `libraryNibNames` method, which returns the names of the library nib files containing the objects you want to integrate into the library window. There are other methods of the `IBPlugin` class you can implement, such as the `label` method, to return information about your plug-in or its configuration. Beyond those basic tasks, however, your plug-in object requires little work to implement.

For more information about the plug-in object and how you use it to manage your plug-in, see "The Plug-in Object" (page 31).

Library Nib Files

Rather than ask your plug-in for the names of the objects it supports, Interface Builder uses nib files to gather that information. This visual approach to reporting your plug-in's contents makes it possible to add new objects to your plug-in quickly and without writing any code. It also makes it possible to support the following features easily:

- You can use a proxy view to provide a different visual representation of your object, if desired.
- You can see how your custom objects and views will look in the library window.
- Interface Builder provides a simple way to specify visual representations for non-visual objects (such as controllers).

For more information on library nib files and how you configure them, see "Configuring the Library Nib Files" (page 31).

Class Description Files

Class description files are property lists that detail the outlets and actions exposed by any custom objects in your plug-in. Because it does not have explicit access to your object code, Interface Builder uses this information to populate the connections inspector and connections panel whenever a user attempts to create a connection to or from your objects.

You create class description files using Xcode and place them in your plug-in's `Resources` directory. When your plug-in is loaded, Interface Builder automatically scans your plug-in bundle for these files and reads in their information. You do not have to tell Interface Builder to do this explicitly.

For more information on how to create class description files for your objects, see "Creating Your Class Description Files" (page 27).

Inspector Objects

The inspector window is where Interface Builder displays the current state of an object's attributes. Interface Builder uses inspector objects to synchronize the contents of the inspector window with the actual properties of the selected objects. An inspector object is an instance of the `IBInspector` class. If your custom objects contain attributes that should be modifiable at design time, you can create a custom inspector object and accompanying user interface to allow the manipulation of those attributes.

In Interface Builder 3.0 and later, attributes are organized by class and displayed in sections inside the inspector window. This approach differs from the one used by previous versions of the software, which presented a panel containing intermingled attributes from various parent classes of the selected object. The class-based organization offers some key advantages over the older approach. Now, your inspector objects need manage only the attributes defined in your custom subclasses, as opposed to all attributes of the class. This approach also makes it possible for Interface Builder to handle multiple selected objects gracefully, allowing the user to modify the attributes that are common to all selected objects. The use of collapsible sections also makes it possible for the user to make more space in the inspector window by hiding unneeded attributes.

For information on how to implement an inspector object and user interface for your custom objects, see "Inspector Objects" (page 39).

The Interface Builder Kit Framework

The Interface Builder Kit framework (`InterfaceBuilderKit.framework`) provides the infrastructure needed to create plug-ins for Interface Builder 3.0 and later. When building a plug-in, you must always link your plug-in bundle against this framework. This framework contains the following support beyond just the key classes (`IBPlugin`, `IBInspector`) that you use to implement your plug-in object:

- The framework defines category methods for both `NSObject` and `NSView` that provide a way for Interface Builder to discover information about your custom objects. Many of these methods provide suitable default implementations but you must override some of them to implement specific features.
- The `IBDocument` class provides support for accessing the data structures of a nib file. You can use instances of this class to locate and manipulate the objects in a nib file, create new connections between objects, and work with pasteboard data.

For more information about the classes of the Interface Builder Kit framework, see *Interface Builder Kit Framework Reference*.

Xcode Support for Interface Builder Plug-ins

Xcode provides a number of templates for creating plug-ins for Interface Builder 3.0. Among these templates are a project template that includes targets for your plug-in bundle and a separate framework for your custom object code. Xcode also includes templates for some of the standard types of files you might add to your plug-in project.

For plug-in development, Xcode also offers improved integration with the Interface Builder environment, providing the ability to create nib files directly from Xcode.

Plug-ins and Threads

Your plug-ins run primarily from the main thread of the Interface Builder application. Because plug-in code is called only as needed by Interface Builder, you should have no need to create additional threads in your plug-in code. For example, you should not use your plug-in to manipulate real data sets or perform computationally-intensive operations.

Plug-ins and Garbage Collection

The Interface Builder application does not use garbage collection for its memory management and your plug-ins should not use garbage collection either. By extension, this also means that the framework that implements your plug-in's objects must be able to run without garbage collection enabled. Because your framework could be linked into a garbage collected application by a client, however, most custom frameworks must be designed as dual-mode frameworks.

A dual-mode framework is one that can operate both with and without garbage collection enabled. To implement a dual-mode framework, you must first configure your framework's Objective-C Garbage Collection build setting so that garbage collection is "supported" and not required. Your framework code then needs to support both memory programming models. In other words, your code must continue to retain and release objects but it must also maintain strong references to objects and abide by other garbage collection guidelines. At runtime, the system essentially "ignores" memory management calls that are not relevant to the current memory mode.

To create a dual-mode framework, you should implement the following guidelines at a minimum. For detailed information about creating a dual-mode framework, see *Garbage Collection Programming Guide*.

- Enable the garbage collection build setting for your framework target.
- Continue to retain and release your objects as you would for a non-garbage collected application. Do not override the `retain`, `release`, and `autorelease` methods in your custom objects, however.
- Make sure you keep strong references to objects in addition to retaining them.
- Do not allocate objects in custom memory zones.
- Initiate collection sweeps only if the `isEnabled` method of `NSGarbageCollector` returns `YES`.
- Implement `dealloc` and `finalize` methods only as necessary. Try to architect your framework to avoid them whenever possible. (For more information about deallocation and garbage collection, see *Garbage Collection Programming Guide*.)

When creating a dual-mode framework, be sure to test your framework in both garbage collected and non garbage collected applications to ensure that it behaves correctly.

Plug-in Deployment Options

How you deploy your Interface Builder plug-in to clients depends heavily on how you deploy your custom controls to those same clients. Apple recommends that you deploy custom controls using a custom framework. A custom framework makes plug-in integration almost trivial for yourself and for the clients of your framework. If you are unable to use a custom framework, however, users can load your plug-in manually into the Interface Builder environment.

Table 1-1 lists the different ways to load a plug-in into interface Builder at runtime.

Table 1-1 Plug-in deployment situations

Situation	Deployment option
At development time…	If you used the standard Xcode template project, you can load your plug-in into Interface Builder by simply building and running your plug-in target. The template project is configured to open Interface Builder automatically and load your plug-in. You can use this option to test your plug-in and make sure its items appear in the library and inspector windows.
If you have a custom framework…	If you are shipping a custom framework with your controls, simply include your Interface Builder plug-in in your framework's `Resources` directory. When clients add your framework to their Xcode projects, Interface Builder automatically loads the associated plug-in for any nib files associated with that project.
If you do not have a custom framework…	You can instruct users to load your plug-in manually using the Interface Builder preferences window. The user can add your plug-in using the provided controls or drag the plug-in bundle into the window. In either case, the plug-in must link against (or contain) the code for your custom controls.

For more information about loading plug-ins, see *Interface Builder User Guide*.

Plug-in Quick Start

If you already have a custom object, creating a basic Interface Builder plug-in for that object takes only a few minutes. This chapter walks you through the creation of a plug-in for a simple `NSButton` subclass. After completing the steps and installing your plug-in, a user should be able to drag your custom button out of the library, add it to a nib file, resize it, reposition it, and save it with the nib file. The user will also be able to customize the basic attributes of the button, since it is derived from the `NSButton` class. The steps for creating the plug-in are as follows:

1. Create your plug-in project using Xcode.
2. Set up your custom button.
3. Configure the library nib file that comes with the project.
4. Build your plug-in and load it into Interface Builder.

This chapter does not cover the steps needed to set up an inspector panel. That information is covered in a later chapter.

Creating and Configuring Your Xcode Project

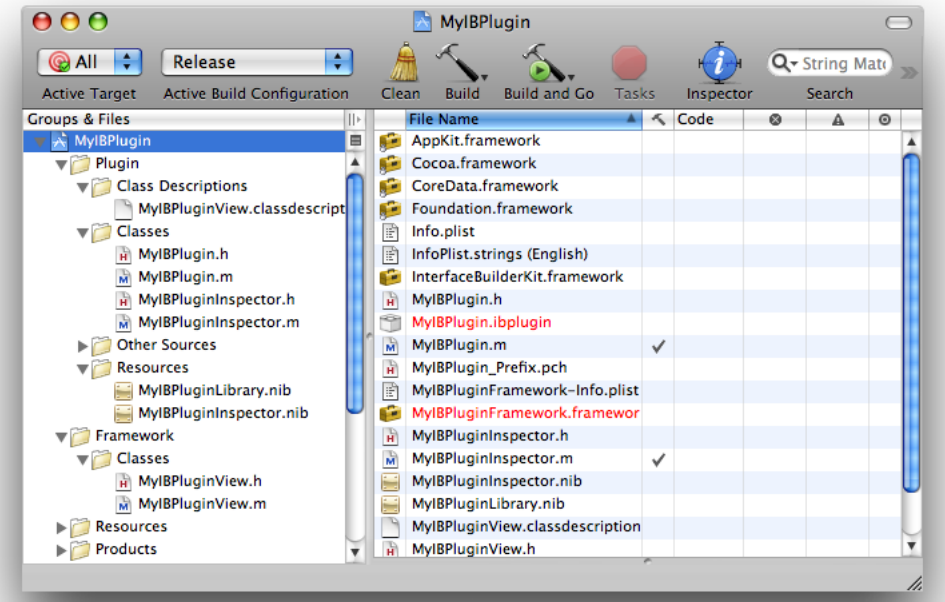
Xcode provides a custom project template for creating an Interface Builder plug-in. To create a new plug-in project, do the following:

1. In Xcode, select File > New Project.
2. In the New Project Assistant, in the Standard Apple Plug-ins section, select the "Interface Builder 3.x Plugin" project type.
3. Enter the project name and location and click Finish.
4. In the new project window, double-click your plug-in target to open the inspector window for that target.
5. In the Properties tab of the inspector window, type a custom bundle identifier name in the Identifier field.

   The bundle identifier differentiates your plug-in from other Interface Builder plug-ins and should contain a string that includes your company name in reverse-DNS format. For more information about bundle identifiers, see *Runtime Configuration Guidelines*.

The project template includes default targets for your plug-in bundle and for a custom framework you can use to encapsulate the code for your custom objects. The code associated with the plug-in target consists of a default subclass of `IBPlugin`(your plug-in's main class) and a basic nib file ready for you to customize.

Figure 2-1 Xcode project for Interface Builder plug-ins



The default plug-in target is configured to build a plug-in bundle with the `.ibplugin` extension. Your plug-in bundle must have this extension for it to be recognized by Interface Builder. If you create a custom Xcode project for your plug-in, you should specify this extension in the Wrapper Extension build setting for your plug-in target.

Set Up Your Custom Button Class

The Xcode project you created includes a framework target that you can use to package your custom objects. Along with this target is a source file you can use for your custom object. This button class should already have a custom name but its parent class is set to `NSView` by default. You need to change the parent class to `NSButton`. Thus, the header file for your new widget should look something like the following:

```
#import <Cocoa/Cocoa.h>
@interface MyIBPluginView : NSButton {
}
@end
```

You do not need to add any additional code to your custom button class.

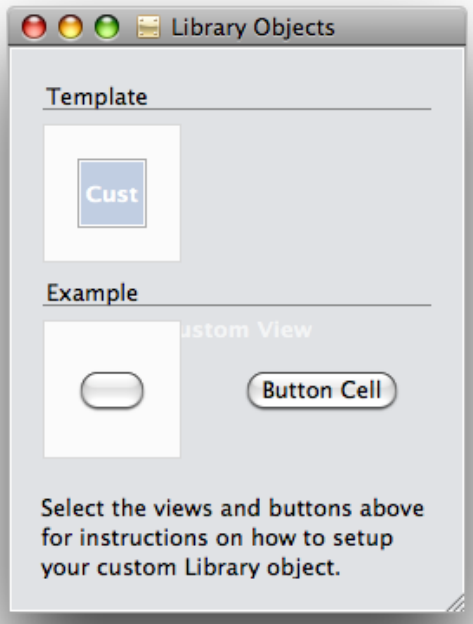
Configuring the Library Nib File

Each new plug-in project contains a library nib file whose name is of the form "*plugin_name*`Library.nib`", where *plugin_name* is the name you give to your Xcode project. Interface Builder looks in this nib file for the custom objects that you want to integrate into the Interface Builder library window. You can use this nib file to integrate both views and non-view objects. Inside the nib file is a default container view called "Library Objects". This view contains one or more library object templates, which are a special type of view used to store the contents of a single library entry.

The library nib file that comes with your Xcode project should already contain two library object template views. One template contains a custom view while the other contains a button and a button cell. Although one of the templates already contains a button, the purpose of the example is to show you how to configure any view. The following steps show you how to configure the template with the custom view.

1. Open the library nib file for your plug-in project in Interface Builder 3.0.
2. In the nib file, double-click the Library Objects view to open it in its own window. Figure 2-2 shows this view, which contains two library object templates (one with a custom view and one with a button).

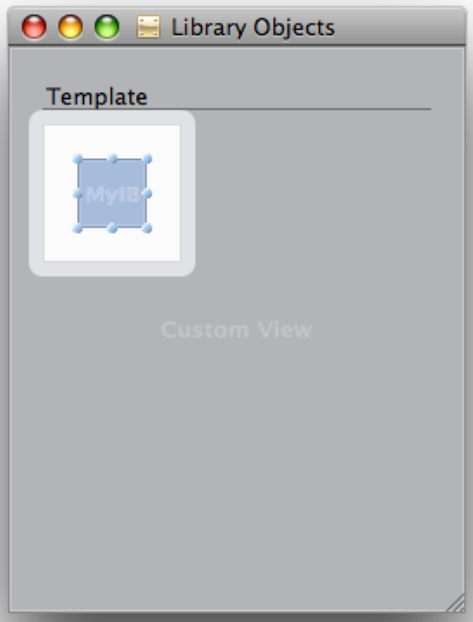
Figure 2-2 Default view in the library nib file



3. Select the bottom library object template (the one with the button) and its surrounding content and delete it.

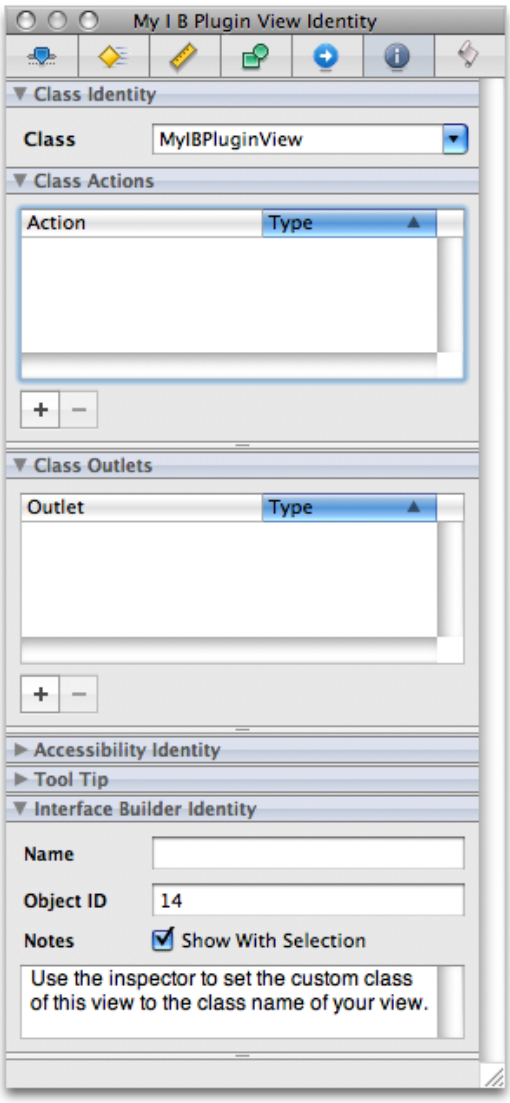
4. Select the custom view inside the remaining library object template. Your view should look similar to the one in Figure 2-3.

Figure 2-3 Removing the unneeded items in the library nib file



5. Open the inspector window and select the identity pane; see Figure 2-4.

Figure 2-4 Identity pane of the inspector window



6. In the Class field of the Inspector window, type the name of your custom button subclass. (From the preceding section, this would be the `MyIBPluginView` class name.)

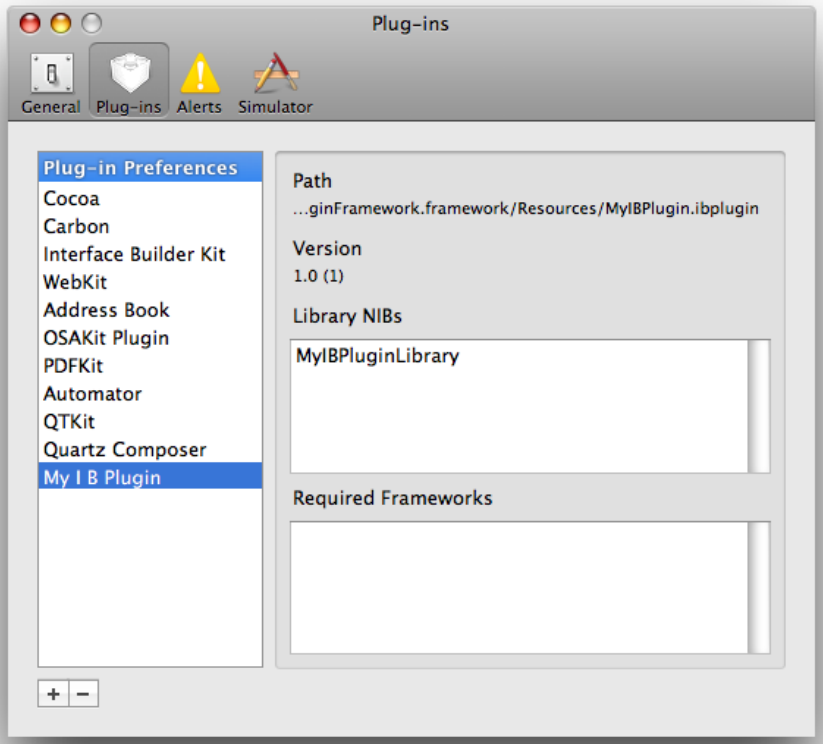
7. Save the nib file.

Once you have saved your nib file, you can proceed to build your plug-in and load it in Interface Builder.

Building and Loading the Plug-in

To build your project, select the All target and press the Build & Go button in the toolbar (or select Build > Build & Run from the menu). The plug-in target comes configured with a dependency on the framework target. Xcode builds your framework in the default build directory and builds your plug-in inside the framework itself. It then launches Interface Builder and loads your plug-in automatically. Your custom button should appear in the library and the preferences window should show your plug-in listing; see Figure 2-5.

Figure 2-5 Interface Builder preferences panel



Xcode builds a plug-in in its corresponding framework directory to facilitate easy distribution of that plug-in to end users. When the user opens a nib file, Interface Builder automatically scans the linked-in frameworks of the associated Xcode project to see if they contain plug-ins. If it finds any, Interface Builder automatically loads those plug-ins to ensure that any custom objects in the nib file can be read. For more information about the runtime integration between Xcode and Interface Builder, see *Interface Builder User Guide*.

If you need to load your plug-in manually for any reason, you can do so from the Interface Builder preferences window. To load your plug-in manually, do the following:

1. Open the Preferences window in Interface Builder.
2. Select the Plug-ins pane.

3. Click the "+" icon at the bottom of the window.

4. Navigate to your plug-in from the sheet and click Open. (Your plug-in should be located inside your framework.)

Preparing Your Custom Objects

As you develop your custom objects, there are some steps you can take to ensure that the integration of those objects into Interface Builder is a smooth process. In many situations, Interface Builder relies on your objects to provide the information needed to initiate certain actions. Wherever possible, Interface Builder relies on standard Cocoa protocols to get the information it needs, but some information, such as your object's inspector class, must be obtained using custom methods.

This chapter provides you with the guidelines you should be following when designing your objects to ensure that they are compatible with Interface Builder later. Many of these guidelines apply to the code for your custom objects and not for the code you put into your plug-in. Some of them are relevant to your plug-in, however, and are called out as such.

Supporting the Basics

Interface Builder uses several standard Cocoa protocols to interact with your objects. You must support these protocols to ensure your objects behave properly in the Interface Builder environment.

- Make sure your custom object is key-value observable (KVO). For more information, see *Key-Value Observing Programming Guide*.
- Make sure your custom object is key-value coding (KVC) compliant. For more information, see *Key-Value Coding Programming Guide*.
- If your custom object has settable attributes, make sure it conforms to the `NSCoding` protocol so that those attributes can be archived and unarchived. For more information, see *Archives and Serializations Programming Guide for Cocoa*.

These protocols make it possible for Interface Builder to offer features such as undo support, pasteboard support, and the automatic refreshing of inspector data for free to your plug-in. You should already be supporting these protocols in your custom code anyway and doing so should not be a significant burden.

If for some reason you cannot support these protocols directly in your object framework, you should at least add comparable support inside your plug-in. Simply create wrapper methods in your plug-in that mimic the behavior of the KVC, KVO, and archiving methods but call through to your object's real accessors behind the scenes. While such a configuration is not optimal, it does provide the overall support Interface Builder needs.

Registering Your Object's Attributes

Before Interface Builder can access any attributes of your custom objects, you must register those attributes in your plug-in code. One of the category methods you must implement for each of your custom objects is the `ibPopulateKeyPaths:` method. Interface Builder calls this method early on, passing it an array of mutable sets, each of which contains a specific type of attribute. To register your object's design-time attributes, you add the key paths for those attributes to these sets.

Listing 3-1 shows a sample implementation of the `ibPopulateKeyPaths:` method for the cell of a control. In this case, the cell has two direct attributes: a title string and a font. The cell also has a pointer to its parent view, which in this case is the control that owns it. Because the parent view can also be manipulated in Interface Builder, it is added to the to-one relationship set.

Listing 3-1 Registering the attributes of a custom object

```
- (void)ibPopulateKeyPaths:(NSMutableDictionary *)keyPaths
{
    // Always call super.
    [super ibPopulateKeyPaths:keyPaths];

    // Add any custom attributes.
    [[keyPaths objectForKey:IBAttributeKeyPaths] addObjectsFromArray:
                [NSArray arrayWithObjects:@"title", @"font", nil]];
    [[keyPaths objectForKey:IBToOneRelationshipKeyPaths] addObjectsFromArray:
                [NSArray arrayWithObjects:@"parentView", nil]];
}
```

Interface Builder monitors the attributes you register using key-value observing notifications. Whenever a change occurs to one of your object's attributes, Interface Builder performs several important tasks. First, it records the change with the current undo manager object. Second, if the change originated in the inspector window, it notifies the object that it should redisplay itself. (Conversely, if the change originated in the window, Interface Builder may notify the appropriate inspector object to refresh itself.)

For a detailed description of the `ibPopulateKeyPaths:` method and how you use it to register your properties, see the method description in *NSObject Interface Builder Kit Additions Reference*.

Additional Design-Time Guidelines

As you design your custom objects, there are some guidelines that, if followed, will make it easier for you to integrate those objects into Interface Builder. Many of these guidelines offer benefits beyond just Interface Builder integration, however, and should be considered in your design plans regardless.

Avoid Cascading Setter Methods

When creating the setter methods for your object's attributes, be sure that each setter method affects only its target attribute. Interface Builder makes extensive use of your object's getter and setter methods to implement undo support. Creating cascading setter methods—that is, setter methods that call additional setter methods—creates extra work to manage the undo stack and may incur a performance penalty.

Use Your Own Accessor Methods

When implementing a custom object, always use the object's setter and getter methods to access attributes instead of the underlying instance variable. Interface Builder implements undo support by monitoring all calls to your setter and getter methods through key-value observing. If you call your own getter and setter methods when responding to user actions, Interface Builder can make sure that those actions are recorded on the undo stack.

Update the Display Inside Setter Methods

If changing an attribute affects the appearance of your view or control, be sure to call the `setNeedsDisplay:` method inside the corresponding setter method. Because Interface Builder manages the undo stack, it may call your setter methods at any time to undo or redo a change. If that happens, your object's setter method may be the only chance it has to refresh the view.

Isolate Interface Builder-Specific Methods

Be aware that in order to make your views and controls interact with it, Interface builder defines additional methods on the `NSObject` and `NSView` classes. Interface Builder uses these methods extensively at runtime to discover information about a particular object. these methods are therefore some of the more important methods for you to implement in your plug-in. Rather than implement the Interface Builder-specific methods in your object's main implementation file, you should always place them in a category that is defined only in your Interface Builder plug-in.

Packaging Your Custom Objects

Once you have your objects designed, you need to package them in a framework that both your client applications and Interface Builder can use. The default Xcode project template for plug-ins includes a target for a custom framework. You can use this target or create one of your own and configure it for your objects. Frameworks are the cleanest way to distribute your custom objects both to your Interface Builder plug-in and to the applications that might use your objects. A framework also lets you maintain a single set of source files and build a single distributable binary package.

When building your framework, be sure to include your Interface Builder plug-in in the `Resources` directory of the framework. When the user loads a nib file, Interface Builder checks for any linked-in frameworks in the associated Xcode project. If those frameworks contain a bundle with an `.ibplugin` extension, Interface Builder automatically loads that plug-in before it opens the nib file.

Creating Your Class Description Files

A class description is a property-list file that enumerates the outlets and actions of your class and also provides other important information to Interface Builder. Class description files reside in your plug-in bundle and should be created from your plug-in's Xcode project. When your plug-in is loaded, Interface Builder searches

its `Resources` directory for any files with the `classdescription` filename extension and reads their contents. It uses the information in those files to configure the connections dialogs and inspector windows with the actions and outlets you specify.

The default Xcode project contains a class description file for you to modify. If your plug-in supports multiple objects, you can add additional class description files as needed. When creating new class description files, be sure to update your plug-in target's Copy Bundle Resources build phase to include the new files.

Listing 3-2 shows a sample class description file for a custom view. This view has a single outlet used to set the object's delegate and two action methods. The first action can be sent by any object but the second must be sent by a view, a condition Interface Builder enforces at design time.

Listing 3-2 Class description for a custom view

```
{
    ClassName = MyCustomView;
    SuperClass = NSView;
    Outlets = {
            delegate = id;
    };
    Actions = {
            "myCustomAction:" = id;
            "myOtherCustomAction:" = NSView;
    };
}
```

You should always provide a class description for each of your custom objects. Each class should have its own class description file and although the exact filename is not important, it is customary to name each class description file after the class it represents. Each class description file should contain the the class name and superclass name information at a minimum. If the class has outlets and actions, you should list those as well; otherwise, you may omit the corresponding sections entirely.

Providing User Documentation for Your Custom Objects

All objects that appear in the library should have some sort of descriptive information describing their purpose and behavior. When creating your plug-in, you associate this information with the library template objects containing your custom objects. Library object templates contain several attributes that not only describe the purpose of your custom object but also specify its location in the library. Table 3-1 lists the basic attributes and how you configure them.

Table 3-1 Library object template attributes

Attribute	Description
`Label`	The name of your custom object. Names should be succinct yet descriptive. Avoid using class names and framework prefixes.

Attribute	Description
`Path`	The path to the group containing the object. If you do not specify path information, all objects appear as children of your plug-in. You can specify additional paths to group objects hierarchically. Paths start with a leading forward slash character and use additional forward slashes to separate hierarchical groups. For example, Cocoa uses the path `/Views & Cells/Buttons` to specify the group for button objects. Interface Builder automatically creates folders in the library window for any path names you specify and nests those folders under your plug-in.
`Summary`	A compact description of your object. Summary text should be no more than 8 to 12 words and should take the form "An <*object*> for <*task*>." Summary descriptions appear in the item pane when descriptions are enabled. They are also used as the tool tip text whenever the user hovers the mouse over the corresponding item.
`Description`	A more complete description of your object. Descriptions should consist of 2 or 3 sentences describing your object's purpose and behavior. These descriptions appear in the documentation pane of the library window when an item is selected.
`Scaling`	The scaling option for your custom object. This determines the technique used to scale your object from its iconic form in the library to its full-fledged form in a nib file. The Image Transformation option causes Interface Builder to scale a bitmap version of your library entry. The View Transformation option causes Interface Builder to send `setFrame:` messages to the actual view to create the animation. The Default mode chooses the mode that looks best for the given object.

The Plug-in Object

The plug-in object is Interface Builder's initial entry point into your plug-in and is represented by the `IBPlugin` class. You must provide a subclass of this object in your plug-in and configure it as the primary class of your plug-in bundle. Your plug-in object has one critical responsibility: provide the list of nib files identifying the objects your plug-in represents. Beyond that, you use the plug-in object to provide support for other features of your plug-in. For example, you use the plug-in object to implement a preferences window or customize objects as they are dragged out of the library window.

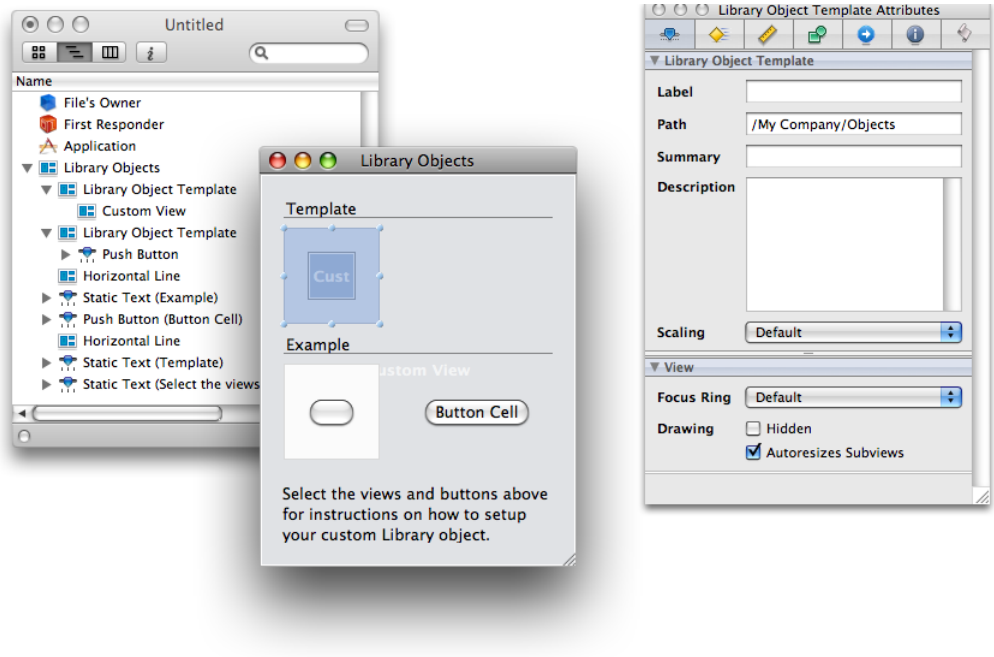
The following sections describe the tasks you can perform from your custom plug-in object. For additional information about the methods of the `IBPlugin` class, see *IBPlugin Class Reference*.

Configuring the Library Nib Files

The main job of the plug-in object is to provide Interface Builder with information about the custom objects it supports. It does this by returning a set of nib file names from its `libraryNibNames` method. These "library nib files" are so named because they contain the objects to be integrated into Interface Builder's library window. Inside the nib file are one or more **library object templates**, which are special containers that hold the visual representation of the objects. Because not all objects have a direct visual representation, library template objects have accommodations for specifying both the real object and a visual proxy of that object.

The best way to understand how the library object templates inside of a library nib file work is to look at an example. Figure 4-1 shows the default nib file that is created whenever you create a plug-in project using the Xcode template. This nib file contains two library object templates. The first of these objects contains a generic `NSView` object. The second contains an `NSButton` object that acts as a proxy for an `NSButtonCell` object. If you were to build and install the plug-in without modifying this nib file, Interface Builder would add two objects to the library window: a generic `NSView` object and a button cell (represented visually by an `NSButton` object). Dragging one of these objects out of the library window would instantiate the corresponding real object (either the `NSView` or `NSButtonCell`) in the user's document. Because they are existing Cocoa objects, the user could then select and configure those objects before saving them with the document.

Figure 4-1 Default library nib file for a sample project



There is no magic to how library object templates work. They are simply views used to identify entries for the library window. The library object template itself is just a container into which you place a single child view. This child view provides the visual appearance of your library entry at runtime. In most cases, this child view is your custom view object. If your object does not descend from `NSView`, or if it is a view but is too large or complex to recognize when scaled to fit the library window, you can instead place an image view in the library object template and use that view to display an iconic representation of your object. You can then use the `representedObject` and `draggedView` outlets of the library object template to associate the real objects to be added to the user's nib file.

The `representedObject` outlet of a library object template points to the object that should be added to a user's nib file in place of the visual representation displayed in the library window. For custom objects that do not descend from `NSView`, you would connect this outlet to an instance of your object that you added to the nib file. Similarly, if you use an image view to draw an iconic version of a view, you would connect this outlet to the actual view that should be added to the user's nib file. If you do not configure this outlet, Interface Builder assumes the view embedded in your library object template is the object that should be added to the user's nib file.

The `draggedView` outlet points to the view that should be used during drag operations from the library window. Dragged views are commonly used to show the user the actual size of views as they are dragged out of the library window. If your plug-in contains a custom object that might be difficult to represent in the limited space available in the library window, you could use a custom icon for the library window and assign the actual view to the `draggedView` outlet. In such a situation, your dragged view would also be the represented object of the entry, unless you specified a different object in the `representedObject` outlet. For more information about setting up dragged views, see "Using a Custom Dragged Vlew" (page 35).

Configuring a Library Object Template

Each new library nib file contains two library object templates already configured with some sample objects. You can reuse these existing templates or delete them and start from scratch. To reuse them, simply replace the contents of those templates with an appropriate view (usually either a generic view or an image view) and configure that view for your custom object.

To add new library object template, do the following in Interface Builder:

1. Open the Library Objects view in your nib file.
2. In the library window, select the Library > IB SDK group. This group contains a single entry, which is a library object template.
3. Drag the library object template from the library window to your Library Objects view. (If you want to resize the library object template, you can do so from the size pane of the inspector window. The width and height of a library object template are typically set to 80 pixels.)
4. Configure the Label, Path, Summary, and Description attributes in the inspector window. For information about what to put in these attributes, see "Providing User Documentation for Your Custom Objects" (page 28).
5. Continue configuring the library object template for your view or object as described in the sections that follow.

Once you have an empty library object template, you can begin configuring it. For each library object template, you should fill in the label, summary, and description fields in the inspector window. These fields provide help information to the user at runtime and are displayed in the library window. The label field provides the basic name of the item while the description field provides detailed information about its purpose. The summary field contains tool tip information and is displayed when the user hovers the mouse over the item.

Configuring a Custom View

To configure a library object template with a custom object that descends from `NSView`, do the following:

1. Locate the generic Custom View object. (It is normally found in the Cocoa > Views & Cells > Layout Views group.)
2. Drag a custom view from the library window and drop it into an empty library object template. (Make sure you drop the custom view so as to make it a child of the template view.)
3. Select the custom view you just dropped and open the inspector window.
4. In the identity pane of the inspector, type the class name of your custom view in the Class field.
5. Save your nib file.

Configuring a Custom Non-View Object

To configure a library object template with a custom object that does not descend from `NSView`, do the following:

1. Configure the object's visual representation:

    a. If you have not already done so, add the image you want to use for your object to your Xcode project. (Make sure the image is included in the Copy Bundle Resources build phase of your plug-in target.)

    b. In Interface Builder, locate the Image Well object in the library window. (It is an instance of the `NSImageView` class and is normally found in the Cocoa > Views & Cells > Inputs & Values group.)

    c. Drag an image well from the library window and drop it into an empty library object template.

    d. Select the image well and open the inspector window.

    e. In the attributes pane of the inspector, select None from the Border popup menu.

    f. In the Image field of the inspector, type the name of the image.

2. Configure the represented object:

    a. In the library window, locate the generic Object. (It is normally found in the Cocoa > Objects & Controllers > Controllers group.)

    b. Drag a generic Object from the library window to your document window, making it a top-level object of your nib file.

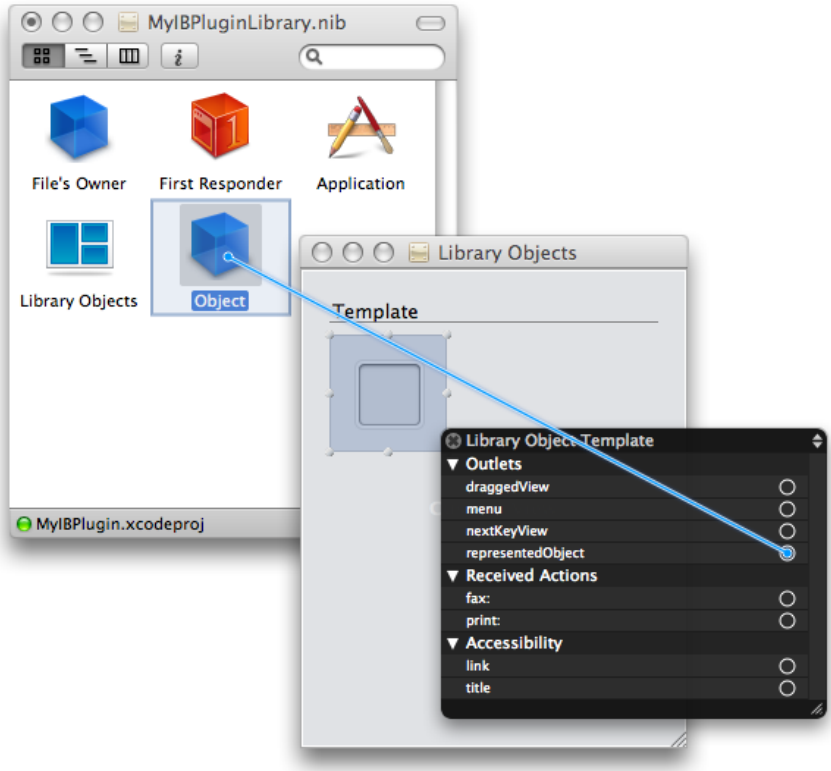
    c. In the identity pane of the inspector, type the name of your custom object in the Class field.

3. Create the connection between the visual representation of your object and the actual object:

    a. Control-click (or right-click) the library object template to display its connections panel.

b. Click and drag from the `representedObject` outlet to the custom object in your document window, as shown in Figure 4-2.

Figure 4-2 Connecting the represented object of a library entry



Using a Custom Dragged Vlew

When the user drags your custom object out of the library window, Interface Builder uses the view inside the library-object template as the default drag image. If you want to use a custom drag image, you can do so by configuring the `draggedView` outlet of the library object template.

A dragged view lets you provide the user with a more appropriately configured view. You might use a dragged view to provide a larger view than the one that appears in the library window. You might also use a dragged view to support a more complex view hierarchy. For example, for a table view item, you might display a table icon in the library window but assign a full-size table view embedded in a scroll view to the `draggedView` outlet. The use of an icon in the library would let you provide a clear visual indicator of what your view represented while the dragged view provides the actual view.

The view you assign to the `draggedView` outlet can be any size you like but must not be embedded inside a library object template. In most cases, you can simply place the view next to your library object template in your nib file, but you can also place it elsewhere in your nib file if doing so is more convenient. Once in your nib file, configure the view the way you want it to appear when dragged from the window.

To assign a dragged view, do the following:

1. Add a new view to your nib file to act as the dragged view. (Size it appropriately.)
2. Control-click (or right-click) the library template object containing your custom object to bring up its connection panel.
3. Click the circle next to the `draggedView` outlet and drag to your dragged view to create a connection.

Creating New Library Nib Files

If you are building a plug-in for a large library of controls, you can use multiple library nib files to organize your plug-in contents. Interface Builder lets you specify any number of library nib files in a single plug-in project, and each nib file can in turn contain multiple library object templates. For example, you could have five library nib files with one library object template each, or you could have one library nib file with five library object templates. Although there is no limit to the number of library nib files your project can include, there is a performance cost to loading many small nib files, so it is recommended that you use a reasonable number of nib files. Creating hundreds of library nib files would not only slow down the loading of your plug-in but would also be a lot of extra work.

You create library nib files the way you would create any nib file in Interface Builder. The new document panel includes an IB Kit tab that when selected shows you the types of nib files you can create for your plug-in. To create a new library nib file, select the Library object and click Choose. Interface Builder creates a new nib file like the one shown in Figure 4-1 (page 32).

After you configure your library nib file and add it to your Xcode project, you need to update the `libraryNibNames` method of your `IBPlugin` subclass. Listing 4-1 shows a sample implementation of this method that returns the names of two custom nib files. You can return as many nib files from your own implementation of this method as you want. Each library nib file can contain one object or multiple objects.

Listing 4-1 The libraryNibNames method

```
- (NSArray *)libraryNibNames
{
    return [NSArray arrayWithObjects:@"myLibraryNibFile1", @"myLibraryNibFile2",
 nil];
}
```

Registering Your Plug-in's Object Frameworks

When you build your plug-in, you link it against the framework containing the code for your custom objects. Thus, when Interface Builder loads your plug-in at design time, it automatically loads your custom object frameworks as well. This is fine for manipulating your objects at design time but causes a problem during simulation. When the user simulates a window, Interface Builder launches an entirely separate process—one that does not load your plug-in code and therefore does not know about your object frameworks. In order to ensure that your objects work properly in the simulator environment, you should override the `requiredFrameworks` method and return the list of frameworks containing your custom objects. Interface Builder passes this list to the simulator environment, which loads the corresponding frameworks as needed to run the simulation.

Listing 4-2 shows a sample implementation of the `requiredFrameworks` method that searches for the desired framework using its bundle identifier string.

Listing 4-2 Returning the required frameworks of a plug-in

```
- (NSArray*)requiredFrameworks
{
    NSBundle*    frameworkBundle = [NSBundle
bundleWithIdentifier:@"com.mycompany.MyFramework"];

    return [NSArray arrayWithObject:frameworkBundle];
}
```

Handling Load and Unload Notifications

Most plug-ins should not require any special initialization, but if yours does, the `IBPlugin` class provides notification methods to let you know when your plug-in is loaded into (or removed from) the Interface Builder environment:

- `didLoad`
- `willUnload`

Most plug-ins should have little need to use either of these methods. If you do use them, do not assume that calls to the `didLoad` method will be balanced by calls to the `willUnoad` method. Although Interface Builder calls the `didLoad` method whenever your plug-in is loaded, it calls the `willUnload` method only when the user explicitly removes your plug-in from the list of plug-ins in the preferences window. Therefore, you should not use your `didLoad` method to acquire resources and the `willLoad` method to release them. You may end up leaking those resources if you do. Instead, release any resources in the `dealloc` or `finalize` method of your plug-in object.

Implementing Plug-in Preferences

When the user selects your plug-in in the preferences window, Interface Builder displays additional information about the plug-in to the right of the plug-in list. By default, Interface Builder shows the list of library nib files and frameworks found in your plug-in but you can use this space to display custom preferences. Doing so is not required, however.

To display a custom preferences view, you must do the following:

1. Create a nib file with an `NSView` object as a top-level object.
2. Set the File's Owner of the nib file to your `IBPlugin` subclass. Your plug-in object should be configured to act as the controller for your preferences view.
3. Add your custom content to the view object and connect any outlets and actions to Files Owner.
4. Override the `preferencesView` method in your `IBPlugin` subclass. In your implementation, load your nib file and return the view object you created.

Your plug-in object should contain all of the outlets, actions, or binding points needed to manage your preferences view. Other objects in your plug-in can access the information in your plug-in object by obtaining the shared plug-in object (using the `sharedInstance` class method of `IBPlugin`) and calling its methods.

Changing your Plug-in Bundle's Principal Class

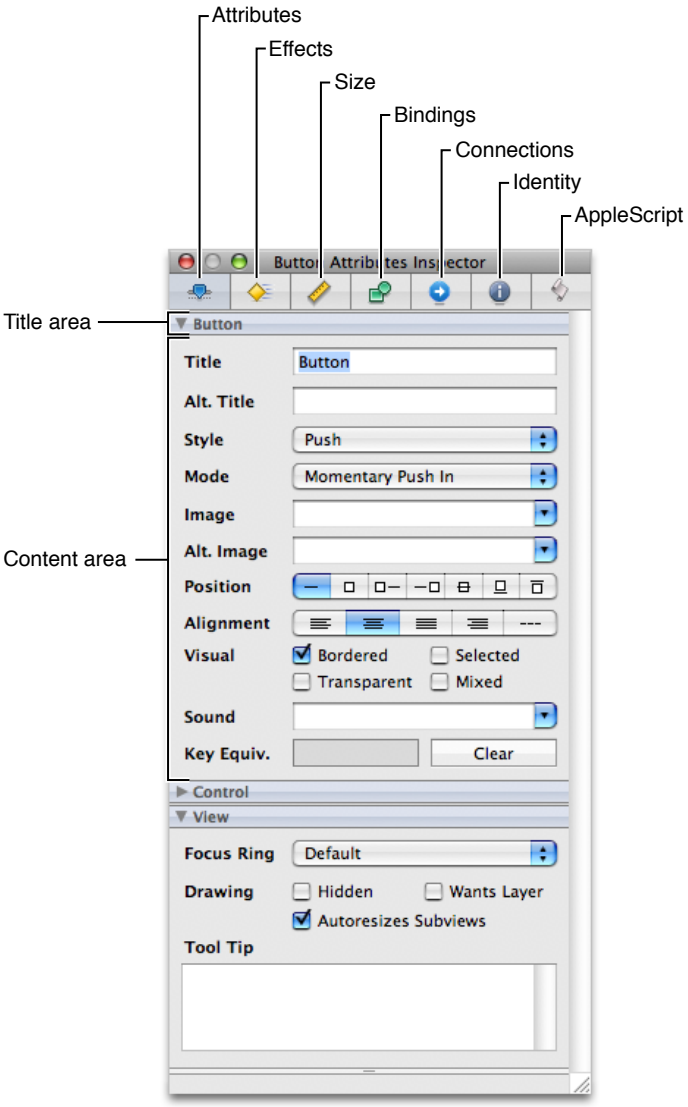
Your custom `IBPlugin` subclass must be the principal class of your plug-in bundle. If you created your project using the Interface Builder 3.x Plugin project template, this information should be configured for you automatically. If you created your project without using the template or renamed your plug-in subclass, you must configure this information manually by doing the following:

1. Open an inspector window for your plug-in target.
2. Select the Properties tab.
3. In the Principal Class field, enter the name of your custom `IBPlugin` subclass.

Inspector Objects

The inspector window in Interface Builder provides the user with access to the attributes of the currently selected objects. The inspector window is divided into several different panes, which are selected using controls at the top of the window (Figure 5-1). The attributes pane is the only pane plug-in developers can customize.

Figure 5-1 The inspector window for Cocoa controls



The attributes pane displays the design-time attributes of the currently selected objects. The attributes themselves are divided up and displayed in "sections", which are collapsible regions consisting of a title bar and content area. The content area of each section contains the attributes associated with a given class in the selected object's lineage. For example, an instance of the `NSButton` class contains sections displaying the `NSView` attributes, `NSControl` attributes, and `NSButton` attributes.

The advantage of sections is that they promote greater editability when multiple objects are selected. When multiple objects are selected, Interface Builder displays all of the inspectors that are common among the selected objects. Thus, if an `NSButton` and `NSTextField` object are selected, the user sees the Control, View, and Object inspectors. This lets the user modify any of the attributes that are common to the objects in the selection.

Each section in the attributes pane is managed by an inspector object. An inspector object ensures that the controls in the section's content view remain synchronized with the attributes of the currently selected objects. When one of your custom objects is selected, Interface Builder queries it for the names of the inspector classes needed to display its attributes. Interface Builder provides inspector classes for all of the standard Cocoa classes so you need to provide inspectors only for those classes you use to implement your custom objects. In addition, an inspector class is needed only if your custom view or object classes have attributes that are configurable at design time. If they do not, you do not need to create an inspector class.

The steps for creating an inspector object are as follows:

1. Define a custom subclass of `IBInspector`.
2. Create a nib file with the user interface of your inspector section.
3. Configure the bindings or write the code needed to synchronize your inspector interface with the currently selected objects.
4. Register your inspector class with Interface Builder.

Creating Your Inspector Class

The `IBInspector` class provides the default controller interface for implementing your custom inspector objects. Custom inspectors are needed only for classes that have custom design-time attributes that you want to be configurable in Interface Builder. If your classes do not expose any public attributes, you do not need to create an inspector class.

Every inspector class has the following responsibilities:

- Provide an interface for viewing and setting attributes.
- Synchronize the controls in its view with the attributes of the current selection.

The `viewNibName` method of `IBInspector` is the preferred way to provide the interface for your inspector class. This method returns the name of the nib file containing your inspector's interface. You can also create your user interface programmatically if you prefer. The steps for creating your user interface are discussed in "Creating Your Inspector's User Interface" (page 41).

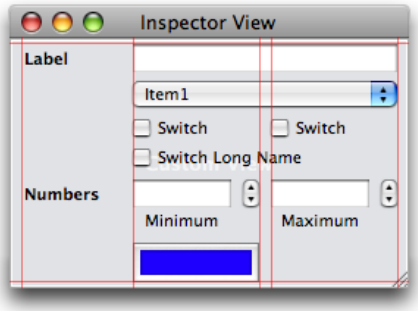
For information on how to synchronize your inspector's interface with the current selection, see "Synchronizing Your Inspector's Interface" (page 42).

Creating Your Inspector's User Interface

There are two ways to create your inspector's user interface: programatically or using a nib file. Using a nib file is by far the simplest way to create your inspector's interface. In fact, if you use Cocoa bindings, it is possible to create your inspector with little or no code at all. Whereas, creating your inspector interface programatically is complicated and requires much more effort and testing to ensure the correct positioning and layout of any controls.

All new plug-in projects in Xcode come with an inspector nib file for you to customize. To create additional inspector nib files, you use Interface Builder. Interface Builder's new document dialog includes an IB Kit tab that contains plug-in related template nib files. From this tab, selecting the Inspector template creates a nib file with the default content view shown in Figure 5-2. This view includes several guides to help you line up your custom controls with the controls found in other inspectors. The view also includes some default controls that you can use for your inspector (or delete if they are not needed). Although you should not change the width of your inspector view, you can (and should) change its height to match the space used by your controls.

Figure 5-2 Default inspector view template



To configure the nib file containing your inspector's user interface, do the following:

1. Open your plug-in project in Xcode. (This ensures that your class header files are accessible in Interface Builder.)
2. Open your project's existing inspector nib file (or create a new one).
3. Add or remove any needed controls in your inspector view.

   Typically, you would provide a single control for each design-time attribute of your object you want to expose. The type of the control would be determined by the type of data represented by the underlying attribute.

   - String values are typically represented by text fields.
   - Numerical values may be displayed in a text field but might also have an optional stepper control to increment or decrement the value.
   - Boolean values are typically represented using check boxes.
   - Enumerated type lists may be represented by radio buttons or pop-up menus.

4. Select the Files Owner proxy object and open the identity pane of the inspector window.

5. Set the class name of File's Owner to your `IBInspector` subclass. (You set this information in the identity pane of the inspector window.)
6. If it is not already connected, connect the `inspectorView` outlet of Files Owner to your inspector view. (This outlet is provided for you by the `IBInspector` class and should be connected already.)
7. Create any other connections or bindings required by your code. (For example, you might want to connect any outlets or actions to their targets.)
8. Save your nib file and add it to your Xcode project (if it has not yet been added).

Nearly all inspector nib files require some additional connections beyond the `inspectorView` outlet of File's Owner. You use these connections to make it possible to synchronize changes as the user changes the current selection and modifies controls in your inspector. Cocoa bindings provide the simplest type of connection by automatically synchronizing the current selection with your inspector's controls. You can also use outlets and actions if you prefer, however. For more information on using both of these techniques, see "Synchronizing Your Inspector's Interface" (page 42).

If you want to create your inspector view programmatically, you can do so by implementing a custom `view` method in your `IBInspector` subclass. In your implementation of this method, you would create the view object you want your inspector to display and configure it as required by your custom object. In addition to overriding this method, you must override the `viewNibName` method and have it return `nil` to prevent Interface Builder from automatically looking for a nib file.

Synchronizing Your Inspector's Interface

Interface Builder relies on your inspector object to coordinate the synchronization of the currently selected objects to your inspector's user interface. Cocoa bindings are the preferred (and simplest) way to synchronize data but you can also use outlets and actions if you prefer. Which technique you choose may also depend on the complexity of your objects and how much logic is required to synchronize them with the inspector controls. Synchronization is required in the following situations:

- The user changes the current selection.
- The user changes the value of one of your inspector's controls.

When multiple objects are selected, the inspector window displays only those inspector sections that are common to all of the selected objects. Your inspector objects must be prepared to handle this situation gracefully by displaying appropriate values in the controls of their user interfaces. Although there are options for situations where handling multiple selected objects is difficult or impossible, you are highly encouraged to design your inspector interface in a way that allows it to display at least some information when multiple objects are selected.

The following sections guide you through the steps to implement the synchronization code for your inspectors. Remember that you can choose to use bindings, outlets and actions, or a combination of both.

Synchronizing Your Interface Using Bindings

Bindings provide a sophisticated and elegant way to synchronize your inspector interface with the currently selected objects. Bindings are especially easy to use with inspectors since the whole point of an inspector is to reflect the values in the current selection—a task for which bindings are well suited.

To establish a binding, select one of the controls in your inspector view and open the inspector window. In the bindings pane, configure your binding to the File's Owner object and use the `inspectedObjectsController.selection` string as the initial part of the model key path. The `inspectedObjectsController` property of the `IBInspector` class is a key-value observable property that returns an `NSArrayController` object with the current selection. Binding through this object provides you with access to the currently selected objects. If your synchronization logic is more complex, you can also include additional controller objects in your nib file and bind to them as needed to implement your logic.

To bind a checkbox to a Boolean value in your custom object, you would do the following:

1. Select the checkbox and open the inspector window.
2. In the bindings pane, expand the Value binding so that you can configure it.
    a. Set the Bind to property to the File's Owner object.
    b. Set the Model Key Path field to a value similar to the following:

        `inspectedObjectsController.selection.`*MyObjectProperty*

        where *MyObjectProperty* is the name of a KVO-compliant attribute in the target object.
3. Configure any other bindings as desired.

For more information about establishing bindings between objects, see *Cocoa Bindings Programming Topics*.

Synchronizing Your Interface Manually

If you prefer use actions and outlets to synchronize your interface, you must do the following to implement your synchronization code:

- Define action methods in your inspector object that synchronize changes in your inspector's controls with the objects in the current selection.
- Implement the `refresh` method of your inspector object to respond to changes in the current selection.

Implementing action methods for your inspector's controls is a relatively straightforward task. When invoked, your action method should get the value from the control that initiated the action and write that value to each of the selected objects. To get the currently selected objects, use the `inspectedObjects` method of `IBInspector`.

Compared to bindings, implementing your `refresh` method involves a little more work, especially to support multiple selected objects. Interface Builder calls your `refresh` method any time the application state changes in a way that might require you to refresh your inspector. These state changes typically involve the user changing the selection but they might also be triggered by the active undo manager or other circumstances.

One of the first things your `refresh` method should do is see how many objects are in the current selection. If only one object is selected, you can simply extract the attribute values from that object and use them to set the state of your inspector's interface. If multiple objects are selected, you need to determine what to display. If all of the values are the same, you should display the common value. If the values are different, you need to convey a multi-selection state in the appropriate controls.

Listing 5-1 shows a `refresh` method for an inspector whose interface contains a single text field, which is assigned to the `titleField` outlet of the inspector object. When a single object is selected, the refresh method simply sets the value of the text field to the value of the object in the array. If multiple objects are selected and all their titles match, this method displays the common title string in the text field. If there is a mismatch in any of the titles, a placeholder string is displayed instead.

Listing 5-1 Handling multiple objects in the refresh method

```
- (void)refresh
{
    NSArray*        objects = [self inspectedObjects];
    NSString*       newTitle;
    NSInteger       numObjects = [objects count];

    if (numObjects == 1)
    {
        newTitle = [[objects objectAtIndex:0] title];
        [titleField setStringValue:newTitle];
    }
    else if (numObjects > 1)
    {
        NSString*    tempString;
        NSInteger    i;
        BOOL        allMatch = YES;

        // See if the titles are all the same.
        newTitle = [[objects objectAtIndex:0] title];
        for (i = 1; i < numObjects; i++)
        {
            tempString = [[objects objectAtIndex:i] title];
            if (![newTitle isEqualToString:tempString])
            {
                allMatch = NO;
                break;
            }
        }

        // Set the value of the text field.
        if (allMatch)
            [titleField setStringValue:newTitle];
        else
        {
            [titleField setStringValue:@""];
            [(NSTextFieldCell*)[titleField cell]
setPlaceholderString:@"<multiple>"];
        }
    }

    [super refresh];
}
```

When implementing a custom `refresh` method, you must invoke `super` at some point in your implementation. Interface Builder uses the `refresh` message to update some of its own internal objects, such as the inspected objects controller. Invoking `super` ensures these objects are updated properly.

Tips for Displaying Attributes for Multiple Selected Objects

When multiple objects are selected, Interface Builder displays only those inspectors that are common among all of the selected objects. Although the types are the same, the attributes of each object may not be the same, however. If the current selection contains multiple objects, your inspector needs to check the values in the objects and determine an appropriate way to convey that information. There are two basic scenarios that can occur:

- All of the objects contain the same value for a given attribute.
- At least one object has a different value for an attribute.

If all of the objects contain the same value, you should display the common value. If the values differ in any way, you need to convey this status to the user somehow. Some controls support the ability to display a mixed state indicator, but others may require that you simply show no value. The following list shows some of the standard controls used in inspectors and how you might use them to display mixed state information:

- Check boxes - display the mixed state setting for the checkbox.
- Text fields - display a placeholder string with the value "*Mixed*" or "<*multiple*>".
- Pop-up buttons and combo boxes - display a blank menu item—that is, a menu item with no text.
- Segmented controls - deselect all segments.
- Radio buttons - deselect all buttons in the group.
- Color wells - display a default color

Disabling Your Inspector During Multiple Selection

If your inspector object cannot inspect a selection with multiple objects, you can tell Interface Builder not to display your inspector when multiple objects are selected. To do this, override the `supportsMultipleObjectInspection` method in your `IBInspector` subclass and return `NO`.

Returning `NO` from the `supportsMultipleObjectInspection` method should be avoided if at all possible. You might use this option, however, when no reasonable alternative exists for reflecting the data of multiple objects. For example, if your inspector displays tabular data or some other complex data that cannot be represented easily for more than one object at a time, you could use this option. Doing so should still be avoided whenever possible, however. Instead, you might consider disabling your table or temporarily replacing it with a text field and the words "Multiple selection".

If it is easy to reflect a multi-object selection for some attributes but not others, it is preferable to disable the one or two problematic controls when multiple objects are selected than disable your entire inspector interface. Disabling the problematic controls lets the user continue to modify other attributes of the object, even when multiple objects are selected.

Registering Your Inspector Objects

Before it can display your inspector interface, Interface Builder needs to know which objects use it. You provide this information by implementing the `ibPopulateAttributeInspectorClasses:` method on your custom object. In your implementation of this method, you add the list of inspector classes that can be used to edit your object to the provided array. Interface Builder then creates inspector objects based on the set of classes you return.

When implementing your `ibPopulateAttributeInspectorClasses:` method, be sure to call `super` before adding any custom classes to the `classes` array. The order in which you add classes to the array defines the resulting order of the inspector sections in the inspector window. The first class in the array appears at the bottom of the inspector window, while the last class appears at the top. This means that you should generally add any inherited inspectors first and add your custom inspectors after that.

Listing 5-2 shows a sample implementation of the `ibPopulateAttributeInspectorClasses:` method for a custom view. This view has a single custom inspector, called `MyInspector`, that it adds to the inherited inspector list.

Listing 5-2 Returning the inspectors for an object

```
@implementation MyCustomView (InspectorIntegration)
- (void)ibPopulateAttributeInspectorClasses:(NSMutableArray *)classes
{
    [super ibPopulateAttributeInspectorClasses:classes];
    [classes addObject:[MyInspector class]];
}
@end
```

For more information about the `ibPopulateAttributeInspectorClasses` method, see *NSObject Interface Builder Kit Additions Reference*.

Advanced Techniques

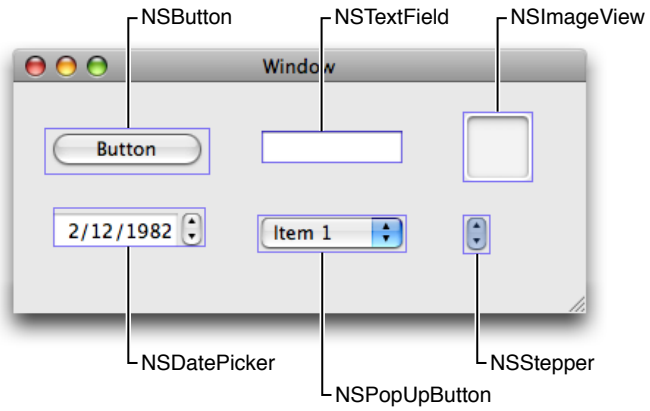
The Interface Builder Kit provides support for tweaking the behavior of your plug-in in many different ways. The following sections provide information about some of the more common ways to modify your plug-in. For information about additional ways to tweak your plug-in, see the objects and methods of *Interface Builder Kit Framework Reference*.

Customizing Your View's Layout

Layout is an important feature of Interface Builder. Users need to be able to position views and controls visually to make sure they line up properly and in accordance with the Aqua style guidelines. To help with alignment, Interface Builder provides dynamic guides that become visible when a view or control is in close proximity to an appropriate boundary of another object.

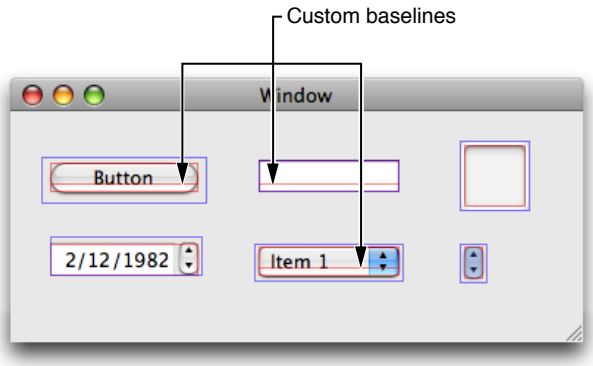
By default, Interface Builder uses a view's frame as the alignment boundaries for the view. This may be appropriate for some views but may not be for others. Many controls have frames that are actually bigger than the main visual component of the control. This is often done to allow room for drawing shadows and other visual effects. Figure 6-1 shows several controls along with their actual frame rectangles, which are in blue. In most cases, the frame rectangles are slightly larger than the drawn portion of the control.

Figure 6-1 Frame boundaries for assorted views and controls.



Rather than using a view's frame rectangle for alignment, it is generally preferable to align views and controls by other means. Most commonly, you would align controls according to the boundaries of their main visual component. For controls with text, you might also want to align the control using the baseline of the text. Figure 6-2 shows the frame rectangles of a set of controls but also shows the inset rectangles and custom baselines of those controls in red. For the user, it makes more sense to use these boundaries to align controls.

Figure 6-2 Inset boundaries and custom baselines



In Interface Builder, you can provide custom baselines and inset values for each of your custom views. You do this by overriding category methods defined on the `NSView` class in the Interface Builder Kit framework. Information about the information to provide in these methods is in the sections that follow.

Specifying Inset Boundaries for a View

To specify inset boundaries for a view, you must override your view's `ibLayoutInset` method. This method is a category method provided by the Interface Builder Kit framework and is described in detail in *NSView Interface Builder Kit Additions Reference*. In your implementation of this method, return an `IBInset` structure containing the inset values needed to arrive at the correct bounding rectangle for your view. The inset values `{0, 0, 0, 0}` result in a rectangle that matches the original frame rectangle of your view. Positive inset values move this rectangle inward towards the center of the view. You should rarely (if ever) specify negative inset values because doing so aligns your view along boundaries that lie outside of its frame rectangle. For example, to specify the inset values for the `NSButton` control shown in Figure 6-2 (page 48), you would write your `ibLayoutInset` method as follows:

```
- (IBInset)ibLayoutInset
{
    IBInset inset = {8, 6, 6, 4};
    return inset;
}
```

For information about the `IBInset` data type, see *Interface Builder Kit Data Types Reference*.

Specifying Custom Baselines

If your view renders text strings, you can specify custom baselines for each of those strings. Custom baselines let the user align your control using its text content instead of its frame or inset boundaries. This kind of alignment is frequently used when aligning text in different types of controls. Baselines are also the more commonly used way to align text-only fields such as labels.

To provide Interface Builder with baseline information, you must override the `ibBaselineCount` and `ibBaselineAtIndex:` methods in your custom view. The `ibBaselineCount` method returns the number of baselines available in your view. The `ibBaselineAtIndex:` method returns the y-axis offset value (relative

to the y-origin of the view's frame rectangle) for one of those baselines. You can change the number of baselines your view supports dynamically if you wish. If you do so, however, the `ibBaselineAtIndex:` method must return a valid value for each index it receives.

For more information about the `ibBaselineCount` and `ibBaselineAtIndex:` methods, see *NSView Interface Builder Kit Additions Reference*.

Implementing a Design-Time Container View

If your view is capable of acting as a container view, you should override the `ibDesignableContentView` method in your custom view. Returning a valid view object from this method tells Interface Builder to treat that view as a container at design time and let the user add child views to it. For most views, you would simply override this method and return `self`, to indicate that the current view is the container, but this need not be the case. If your main view is not the view you want to use as the container, you could return an associated subview. For example, a scroll view contains a clip view, which in turn contains the document view to be scrolled. In that situation, the scroll view would return its clip view object as the container view.

Although container views allow their children to be moved and resized freely for the most part, you can alter this behavior by overriding the `ibIsChildViewUserMovable:` and `ibIsChildViewUserSizable:` methods in your container view. You should only need to override these methods if allowing the user to move or resize child views would cause problems for your view. For example, scroll views do not allow users to move or resize their contained scrollers. The position and size of the scrollers remains fixed based on the scroll view's frame and cannot be changed by the user at design time.

For more information about these methods see *NSObject Interface Builder Kit Additions Reference* and *NSView Interface Builder Kit Additions Reference*.

Exposing Embedded Child Objects

In each document, Interface Builder distinguishes between objects that can be selected and inspected by the user and those that are simply there in a supporting role. Objects that can be selected and inspected are considered "first-class" objects and are the only objects users actually see in a document. Supporting objects are present in the nib file but cannot be configured by the user and are generally not visible. Both first-class objects and supporting objects are saved as part of the resulting nib file.

For each library entry, Interface Builder exposes only the represented object of the entry as a first-class object by default. (For more about the structure of library entries, see "Configuring a Library Object Template.") The actual implementation of a represented object can have any number of associated child objects, however, all of which are relegated to supporting roles by default. To change child objects into first-class objects, you expose them to Interface Builder by overriding the `ibDefaultChildren` method of their nearest parent object..

The `ibDefaultChildren` method simply returns a list of objects to expose as first-class children of the receiver. For each exposed child, Interface Builder calls the child's own `ibDefaultChildren` method to give that child a chance to expose its own children. When exposing child objects, give careful consideration to which objects you want to expose. Views in a view hierarchy commonly expose configurable child views. They may also expose cell objects, which although not views do contain configurable attributes, outlets, and actions. It is rare for non-view objects to expose child objects—that is, you typically would not expose any child objects from a controller object.

The sections that follow discuss some of the ways a parent view can manipulate the information surrounding its child objects. For more information about implementing the `ibDefaultChildren` method or any of the methods discussed in the following sections, see *NSObject Interface Builder Kit Additions Reference*.

Controlling the Size Attributes of Embedded Child Views

The position and size attributes of a view are controlled by its parent (or container) view. The resizing and repositioning of child views within their parent view is handled automatically by the default editor object of the parent view. To change this behavior, simply override the `isChildViewUserMovable:` or `isChildViewUserSizable:` method to disable the behavior for the specified view.

Controlling the Selection of Child Objects

Normally, when the user clicks in a window at design time, Interface Builder tries to select the object that is directly under the mouse. For some complex view hierarchies, this might involve selecting an object deep inside the view hierarchy, which may not always be useful. If you expect users to use your view hierarchy primarily as a group, you might want the first click to select the top-level object, making it easier to drag or resize the entire group of views. To make sure the initial mouse click selects your top-level view, you can override the `ibIsChildInitiallySelectable:` method and return `NO`. Overriding this method does not prevent the child from being selected, it just forces the user to select the parent view first and then select the child afterwards.

Returning Geometry Information for Non-View Objects

Although Interface Builder knows how to identify views at design time, it needs help when it comes to your custom non-view objects. If you use non-view objects in the view hierarchy, you should override the `ibRectForChild:inWindowController:` and `ibObjectAtLocation:inWindowController:` methods to provide Interface Builder with the geometry information it needs for your custom objects.

Configuring Objects at Design Time

Interface Builder sends notifications to your custom objects whenever they are added to or removed from a user's document. You can implement the `ibAwakeInDesignableDocument:`, `ibDidAddToDesignableDocument:` and `ibDidRemoveFromDesignableDocument:` methods in your custom object to receive these notifications. You might use these notifications to perform additional configuration steps involving other objects in the document. For example, you could create or remove connections between your custom objects and the File's Owner, First Responder, or Application object of the document.

For more information about these notification methods, see *NSObject Interface Builder Kit Additions Reference*.

Document Revision History

This table describes the changes to *Interface Builder Plug-In Programming Guide*.

Date	Notes
2007-07-18	New document describing the process for creating custom Interface Builder palettes.

Document Revision History