Interface Builder User Guide

Tools & Languages: IDEs

2010-07-12

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

Interface Builder is a visual design tool you use to create the user interfaces of your iOS and Mac OS X applications. Using the graphical environment of Interface Builder, you assemble windows, views, controls, menus, and other elements from a library of configurable objects. You arrange these items, set their attributes, establish connections between them, and then save them in a special type of resource file, called a **nib file**. (The term "nib" is historical and is an acronym for "NextSTEP Interface Builder.") A nib file stores your objects, including their configuration and layout information, in a format that at runtime can be used to recreate the actual objects.

Who Should Read This Document?

This document discusses the features of Interface Builder 3.2 and also describes the nib file design process. User interface designers should read this document to learn how to use Interface Builder to create the desired look of their application. Programmers should also read this document to understand what program-level information needs to be created in Xcode.

Organization of This Document

This document contains the following chapters:

- "Interface Builder Quick Start" (page 17) gives you a quick tour of Interface Builder, including a hands-on tutorial.
- "Interface Builder Basic Concepts" (page 33) provides an overview of Interface Builder and the role of nib files in your applications.
- "Xcode Integration" (page 139) provides an overview of how nib files integrate with your Xcode projects.
- "Nib File Management" (page 47) shows you how to create and save nib files and how to work with them in your projects.
- "Nib Objects" (page 55) describes the types of objects you can add to nib files and when you might want to do so.
- "Interface Layout" (page 89) describes techniques for organizing window and view hierarchies and for adjusting the layout of individual objects.
- "Object Attributes" (page 105) describes the techniques for configuring the objects in your nib file with custom attributes.
- "Connections and Bindings" (page 119) describes the role of connections in nib files and how to configure outlets, actions, and bindings.
- "Testing and Validation" (page 149) describes the tools available for testing your nib file contents and ensuring your nib file works properly with your Xcode project.

- "Localization" (page 153) describes the tools and processes for localizing the contents of a nib file.
- "Interface Builder Customization" (page 159) describes the ways in which you can change the Interface Builder environment to suit your personal tastes.
- "Interface Builder Gesture Guide" (page 169) summarizes the keyboard and mouse actions you can use to make connections, execute commands, and so on.

The Glossary at the end contains a list of Interface Builder terms and their definitions.

Platform and Language Support

Although sometimes thought of as a tool for designing Cocoa application interfaces, Interface Builder lets you create nib files for both Objective-C and C-based applications in iOS and Mac OS X. All developers can take advantage of Interface Builder's visual environment for assembling, laying out, and configuring windows and menus. If you are developing applications based on the AppKit or UIKit frameworks, you can also use Interface Builder to interconnect the objects in your nib file and application to facilitate the passing of messages. These connections reduce the amount of code that you need to write and, because they are made using Interface Builder, are easy to change later.

Interface Builder's support for Cocoa in Mac OS X extends beyond just the Objective-C language. You can also use the Cocoa scripting bridge support to create interfaces for applications written using the Ruby or Python scripting languages. For more information about Interface Builder's support for these languages, see "Interface Builder Documents" (page 33).

Most of the chapters in this book cover features that are common to the development of all types of applications. Chapters and sections that are specific to one development environment or another are called out as such.

Getting Interface Builder

Apple provides a comprehensive suite of developer tools (including Interface Builder) for creating iOS and Mac OS X software. The Xcode tools include applications to help you design, create, debug, and optimize your software. This suite also includes header files, sample code, and documentation for Apple technologies. You can download Xcode from the members area of the Apple Developer Connection (ADC) website (http://connect.apple.com/). Registration is required but free.

Reporting Bugs

If you encounter bugs in Apple software or documentation, you are encouraged to report them to Apple. You can also file enhancement requests to indicate features you would like to see in future revisions of a product or document. To file bugs or enhancement requests, go to the Bug Reporting page of the ADC website, which is at the following URL:

http://developer.apple.com/bugreporter/

You must have a valid ADC login name and password to file bugs. You can obtain a login name for free by following the instructions found on the Bug Reporting page.

See Also

For information on how to use nib files at runtime, including how to load them from your code, see *Resource Programming Guide*.

Introduction

Interface Builder Quick Start

Interface Builder is Apple's graphical editor for designing and testing application user interfaces. This chapter provides a quick tour of Interface Builder, including a hands-on tutorial. If you have used Interface Builder before, you may want to proceed to the next chapter, "Interface Builder Basic Concepts" (page 33).

Note: Interface Builder is included in the Xcode toolset. You can download the latest version of Xcode from the Apple Developer website (http://developer.apple.com/). Registration is required but free.

Opening Interface Builder

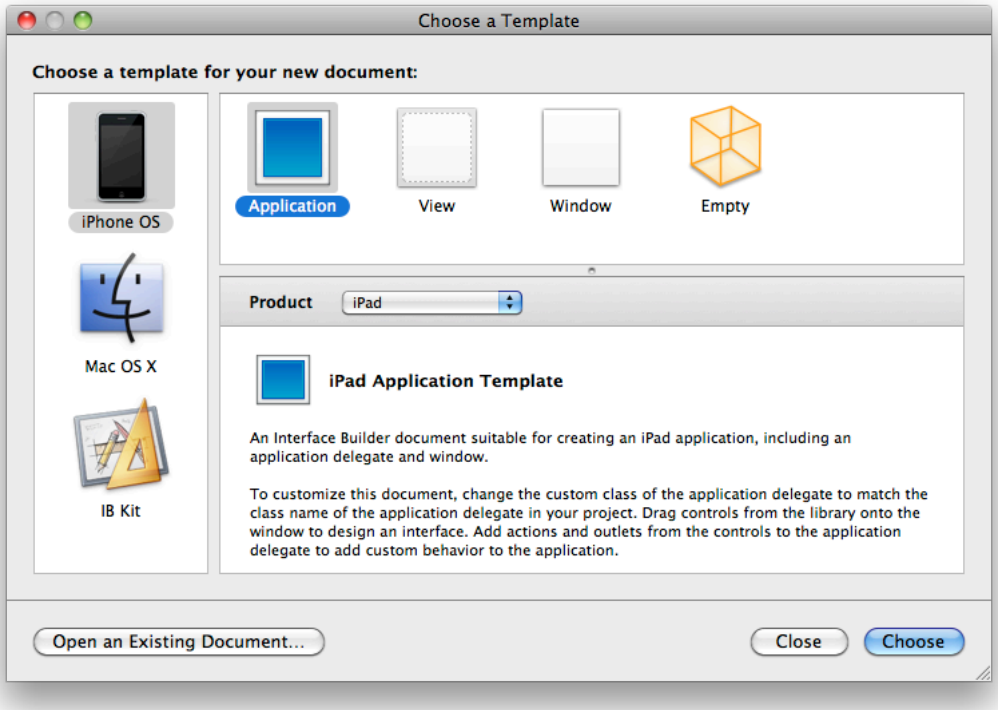
There are several ways to open Interface Builder:

- You can open Interface Builder by double-clicking its application icon in the Finder. The Interface Builder application is located in *<Xcode>*`/Applications`, where *<Xcode>* is the root directory of your Xcode installation. (The default root directory for an Xcode installation is the `/Developer` directory.)
- You can open Interface Builder by double-clicking an Interface Builder document in the Finder. Interface Builder documents are files with the extension `.nib` or `.xib`. These documents are often referred to as *nib files*.
- You can open Interface Builder by double-clicking the name of an Interface Builder document in an Xcode project window.

Creating an Interface Builder Document

When you're developing a new application, you generally start by creating an Xcode project using a project template. New projects come with a basic set of Interface Builder files (also called *documents*). As you continue to develop your project, you'll most likely need to create new Interface Builder documents. You can do that in the Xcode New File dialog, or in Interface Builder by choosing File > New. Figure 1-1 illustrates Interface Builder's new document dialog on a system being used for iOS or Mac OS X development.

Figure 1-1 New document dialog



For each new document you create, Interface Builder prompts you to select a starting template. These templates define the initial set of objects to use in your document. Interface Builder provides several different templates, listed in Table 1-1, each geared toward a different goal. You can use the Empty template if you want to add a specific set of objects to your document manually. For any of the templates, you can also remove objects you do not want.

Table 1-1 Interface Builder document templates

Platform	Product	Template	Description
iOS	iPhone	Application	Creates a document you can use to design an interface for a Cocoa Touch application. The document includes a window.
		View	Creates a document you can use to design a view for a Cocoa Touch application.
		Window	Creates a document you can use to design a window for a Cocoa Touch application.
		Empty	Creates a document to which you can add your own interface objects. To learn how to select and add these objects, see "Workflow Tools" (page 35).
iOS	iPad	Application	Creates a document suitable for creating an iPad application, including an application delegate and window.
		View	Creates a document you can use to design a view for an iPad application.

Platform	Product	Template	Description
		Window	Creates a document you can use to design a window for an iPad application.
		Empty	Creates a document to which you can add your own interface objects. To learn how to select and add these objects, see "Workflow Tools" (page 35).
Mac OS X	Cocoa	Application	Creates a document you can use to design an interface for a Cocoa application. The document includes a menu bar and a window.
		Main Menu	Creates a document you can use to design a menu bar for a Cocoa application.
		View	Creates a document you can use to design a view for a Cocoa application.
		Window	Creates a document you can use to design a window for a Cocoa application.
		Empty	Creates a document to which you can add your own interface objects. To learn how to select and add these objects, see "Workflow Tools" (page 35).
Mac OS X	Carbon	Application	Creates a document you can use to design an interface for a Carbon application. The document includes a menu bar and a window.
		Dialog	Creates a document you can use to design a dialog for a Carbon application.
		Main Menu	Creates a document you can use to design a menu bar for a Carbon application.
		Window	Creates a document you can use to design a window for a Carbon application.
		Empty	Creates a document to which you can add your own interface objects. To learn how to select and add these objects, see "Workflow Tools" (page 35).
Interface Builder Kit	-	Inspector	Creates a document you can use to design an inspector for an Interface Builder plug-in.
	-	Library	Creates a document you can use to design library components for an Interface Builder plug-in.

The Document Window and Workflow Tools

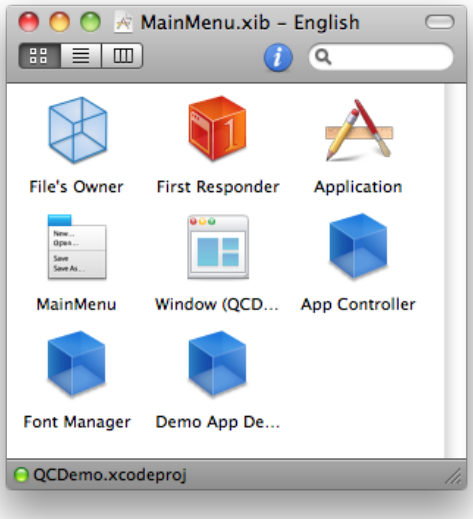
Interface Builder provides several windows to allow you to display and modify the objects in your user interface. This section introduces you to four of these windows. You'll learn more about them in the next chapter, "Interface Builder Basic Concepts" (page 33).

The Document Window

Each Interface Builder document stores information about one or more objects you want to create at runtime in your application. Most of these objects correspond to elements displayed on the screen, such as windows, views, controls, and menus. Some objects do not correspond to displayed elements, such as the controller objects your application uses to manage its windows and views.

The document window lists the objects in a document. Figure 1-2 shows an example of a document window.

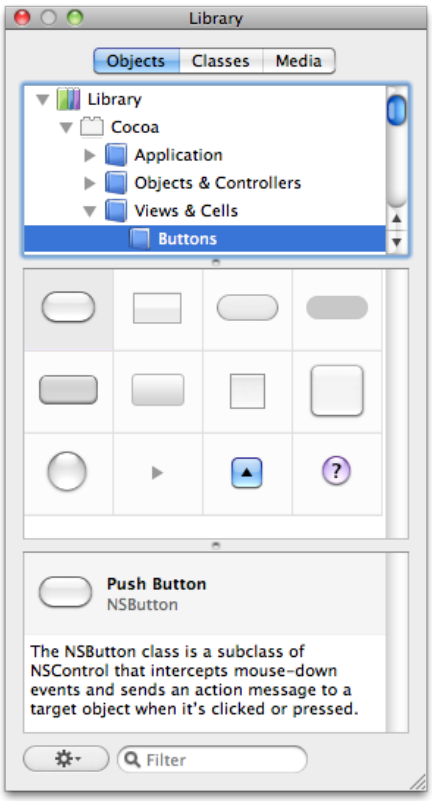
Figure 1-2 The document window



The Library Window

The Library window contains the objects and resources you add to your Interface Builder documents. Figure 1-3 shows the Library window with a set of Cocoa button objects. You can drag a button object onto a design surface such as a window or view.

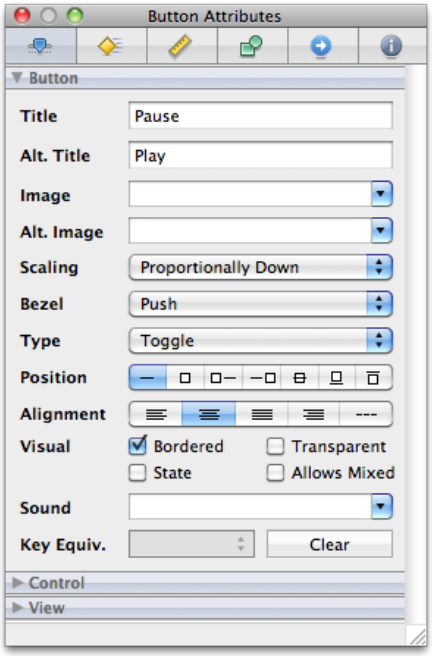
Figure 1-3 The Library window



The Inspector Window

The inspector window makes it easy to display and adjust the settings for the currently selected objects. You use the mode icons along the top of the window to select a pane and display the associated settings. Figure 1-4 shows the Attributes pane for a Cocoa button.

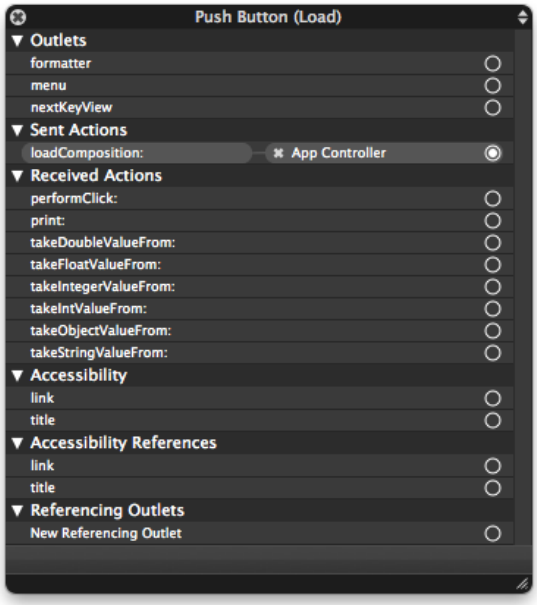
Figure 1-4 The inspector window



The Connections Panel

A connection is a way to associate interface elements with source code. For documents used on the Cocoa and Cocoa Touch platforms, the connections panel is a quick way to create and manage the connections for a particular object.

Figure 1-5 The connections panel

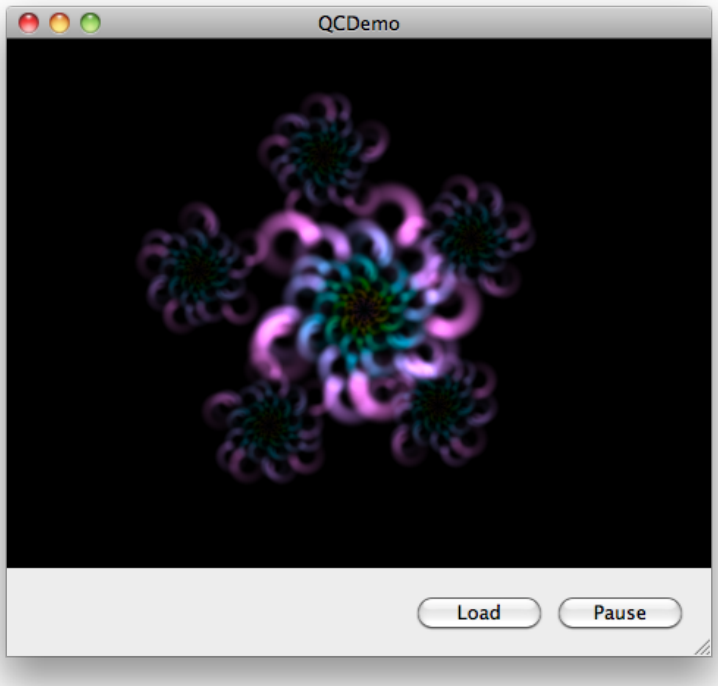


Example: Building a Quartz Composer Viewer

Quartz Composer (QC) is a development tool for processing and rendering graphical data. Its visual programming environment lets you develop graphic processing modules, called *compositions*, without writing a single line of code.

In this tutorial, you'll create a simple application for viewing QC compositions in Mac OS X. The name of this application is QCDemo. The QCDemo window contains a view to display a composition, a button to load a new composition, and a button to toggle between play and pause. Here's what the QCDemo window looks like:

Figure 1-6 QCDemo window



Note: To complete this tutorial, you'll need to install Mac OS X v10.6 or later and Xcode 3.2 or later.

Creating the Viewer

The first task is to create a simple Quartz Composer viewer.

1. Open Xcode and create a project using the Mac OS X Cocoa application template. Name the project QCDemo.

2. Add the Quartz framework to the project's target.

a. In the Xcode Groups & Files list, select the QCDemo target.

b. Press Command-I to display the Target Info window.

c. Click General.

d. Under linked libraries, click the plus (+) button. A dialog appears with a list of libraries.

e. Choose `Quartz.framework` from the list and click the Add button.

f. Close the Target Info window.

3. Open The QCDemo project's Interface Builder document.

a. In the project's Groups & Files list, open the Resources group.

b. Find the file `MainMenu.xib`.

c. Double-click `MainMenu.xib` to open the document.

4. Set the size attributes of the QCDemo window.

a. In the document window, select the window object.

b. Press Command-3 to open the Size pane of the inspector window.

c. Set the window's width and height to 480 x 420.

d. Select the Minimum Size option.

e. Set the minimum size to the current size by clicking the Use Current button.

5. Add a Quartz Composer view to the QCDemo window.

a. Press Command-Shift-L to open the Library window.

b. Click Objects and enter `quartz` in the search field at the bottom of the window.

c. Drag the Quartz Composer view object from the Library window to the QCDemo window.

6. Set the size attributes of the Quartz Composer view.

a. In the QCDemo window, select the Quartz Composer view object.

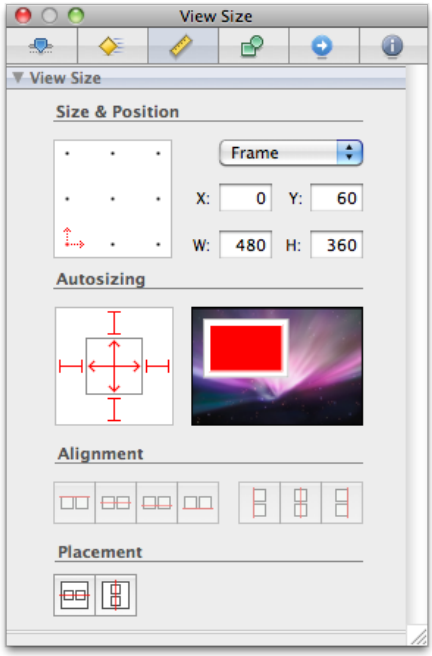
b. Press Command-3 to open the Size pane of the inspector window.

c. Set the view's position X and Y to 0 x 60.

d. Set the view's width and height to 480 x 360.

e. Set the view's autosizing control as shown in Figure 1-7. To learn more about this control, see "Setting a View's Autosizing Behavior" (page 94).

Figure 1-7 Autosizing the Quartz Composer view



7. Select an initial composition for the Quartz Composer view.

   a. In the QCDemo window, select the Quartz Composer view object.

   b. Press Command-1 to open the Attributes pane of the inspector window.

   c. Click the Load button.

   d. Select the composition at the following location:

   ```
   /System/Library/Screen Savers/Shell.qtz
   ```

8. Save the Interface Builder document.

9. Run a simulation of the QCDemo user interface.

   a. Press Command-R to start the simulation and display the composition.

   b. Try resizing the QCDemo window and observe what happens to the composition.

   c. Press Command-Q to quit the simulation.

10. In Xcode, build and run the QCDemo application.

    a. Click Build and Run.

b. Verify that the application runs correctly.

c. Press Command-Q or click Stop to quit the application.

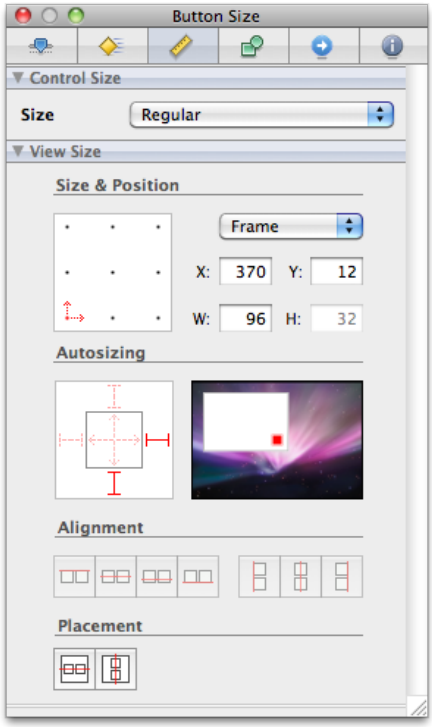
Adding the Pause/Play Button

The next task is to add a button that toggles between pause and play.

1. Add a push button to the user interface.
    a. In the Interface Builder Library window, click Objects.
    b. Type `push` in the search field.
    c. Drag a push button from the Library window to the open area at the bottom of the QCDemo window.
    d. Use the horizontal and vertical guides to position the button in the lower-right corner of the window.
2. Set the attributes of the push button.
    a. Press Command-1 to open the Attributes pane.
    b. Set the button's title to Pause and its alternate title to Play.
    c. Change the button type to Toggle.
    d. The settings in the Attributes pane should look like Figure 1-4 (page 22).

3. In the Size pane, set the button's autosizing control as shown in Figure 1-8.

Figure 1-8 Autosizing the pause/play button



4. Connect the button to the QC view's `play:` action.
    a. Hold down the Control key.
    b. Drag from the Pause/Play button to the QC view.
    c. Select the `play:` action from the list that appears.

5. Save the Interface Builder document.

6. Run a simulation of the user interface.
    a. Press Command-R to start the simulation.
    b. Try out the button and verify its behavior.
    c. Resize the QCDemo window and make sure the button retains its correct position.
    d. Press Command-Q to quit the simulation.

7. In Xcode, build and run the QCDemo application.
    a. Click Build and Run.
    b. Verify that the application runs correctly.

c. Press Command-Q or click Stop to quit the application.

Creating the AppController Class

The third task is to create a class in the Xcode project. The purpose of this class is to load QC compositions.

1. Create the source files for a new objective-C class.
   a. In the Xcode project window, select the Classes group.
   b. Choose File > New File.
   c. In the New File dialog, choose the Mac OS X Cocoa Objective-C class template and click Next.
   d. Name the file `AppController.m` and click Finish. Xcode creates the files `AppController.m` and `AppController.h`.
2. Add the code in Listing 1-1 to the `AppController.h` source file.

Listing 1-1 AppController class interface

```
#import <Cocoa/Cocoa.h>
#import <Quartz/Quartz.h>

@interface AppController : NSObject
{
    IBOutlet NSWindow* qcWindow;
    IBOutlet QCView* qcView;
}

- (IBAction) loadComposition:(id)sender;

@end
```

3. Save the `AppController.h` source file.
4. Add the code in Listing 1-2 to the `AppController.m` source file.

Listing 1-2 AppController class implementation

```
#import "AppController.h"

@implementation AppController

- (IBAction) loadComposition:(id)sender
{
    void (^handler)(NSInteger);

    NSOpenPanel *panel = [NSOpenPanel openPanel];

    [panel setAllowedFileTypes:[NSArray arrayWithObjects: @"qtz", nil]];
```

```
        handler = ^(NSInteger result) {
            if (result == NSFileHandlingPanelOKButton) {
                NSString *filePath = [[[panel URLs] objectAtIndex:0] path];
                if (![qcView loadCompositionFromFile:filePath]) {
                    NSLog(@"Could not load composition");
                }
            }
        };

        [panel beginSheetModalForWindow:qcWindow completionHandler:handler];
    }

    @end
```

5. Save the `AppController.m` source file.
6. Build the QCDemo application (to verify that the code you just added is free of errors.)
    a. In Xcode, click Build or press Command-B.
    b. Verify that the message "Build succeeded" appears at the bottom of the project window.

Adding the AppController Class

The next task is to add an instance of the new class to the Interface Builder document and connect the class outlets.

1. Create an instance of the AppController class.
    a. In the Interface Builder Library window, click Classes.
    b. Locate the AppController class.
    c. Drag this class into the document window to create an instance named App Controller.
2. Connect the App Controller's `qcWindow` outlet to the design window.
    a. Hold down the Control key.
    b. In the document window, drag from the App Controller to the Window object.
    c. Select the `qcWindow` outlet from the list that appears.
3. Connect the App Controller's `qcView` outlet to the QC view in the design window.
    a. Hold down the Control key.
    b. Drag from the App Controller to the QC view in the design window.
    c. Select the `qcView` outlet from the list that appears.

Adding the Load Button

The final task is to add a button that allows the user to select and load another composition.

1. Add a second push button to the user interface.
    a. In the Library window, click Objects.
    b. Type `push` in the search field.
    c. Drag a push button from the Library window to the open area at the bottom of the QCDemo window.
    d. Position the new button to the left of the Pause/Play button.
2. In the Attributes pane of the inspector, set the button's title to Load.
3. In the Size pane of the inspector, set the button's autosizing control as shown in Figure 1-8 (page 27).
4. Connect the button to the App Controller's `loadComposition:` action.
    a. Hold down the Control key.
    b. Drag from the Load button to the App Controller.
    c. Select the `loadComposition:` action from the list that appears.
5. Save the Interface Builder document.
6. Run a simulation of the user interface.
    a. Press Command-R to start the simulation.
    b. Verify that the QCDemo window looks like the window shown in Figure 1-6 (page 23).
    c. Press Command-Q to quit the simulation.
7. In Xcode, build and run the QCDemo application.
    a. Click Build and Run. The QCDemo window appears.
    b. Click Load. A sheet dialog appears.
    c. Use the dialog to load another Quartz Composer composition. For example, load:

        ```
        /System/Library/Screen Savers/Arabesque.qtz
        ```

    d. Press Command-Q to quit the application.

What's Next?

Now you can begin to explore the Interface Builder application in more detail.

Interface Builder Basic Concepts

Interface Builder is the application you use to assemble the visual components (such as windows and menus) of your application. In Interface Builder, you assemble your windows and menus by dragging preconfigured objects into an Interface Builder document. You can reposition those objects and change their attributes as needed to achieve the desired look for your interface. For some application types, you can even create explicit relationships between those objects, which ultimately result in the creation of links between objects at runtime.

This chapter provides a high-level overview of the concepts surrounding the Interface Builder application, including an introduction to nib files (the Interface Builder document type) and the basic features of the Interface Builder application. If you have never used Interface Builder before, you should read this chapter at least once to familiarize yourself with these fundamental concepts. While reading this chapter, you may also want to have the Interface Builder application open so that you can explore features as you read about them.

Note: Interface Builder is included in the Xcode toolset. You can download the latest version of Xcode from the members area of the Apple Developer Connection (ADC) website (http://connect.apple.com/). Registration is required but free.

Interface Builder Documents

Each Interface Builder document stores information about one or more objects you want to create at runtime in your application. Most of these objects are visual in nature. Typical objects include windows, views, controls, and menus. Some application types let you specify non-visual objects as well, such as the controller objects your application uses to manage its windows and views.

Figure 2-1 shows a sample Interface Builder document along with some of the objects it contains. The toolbar includes several controls to help you manage the contents of the document and the status bar contains information about the document's association with an Xcode project (if any).

Figure 2-1 Interface Builder document window

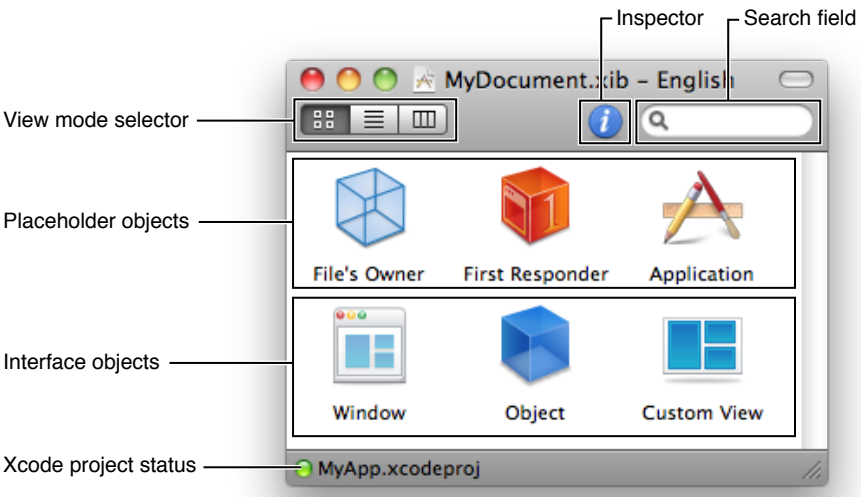


Table 2-1 lists the key controls in the Interface Builder document's toolbar and how you use them to manage the document's contents.

Table 2-1 Interface Builder document controls

Control	Description
View mode selector	Lets you pick between Icon, Outline, and Browser modes for viewing your document contents. Icon mode shows the key objects of your nib file only. Outline mode displays the objects in a hierarchical tree that reflects the parent-child relationships between the objects. Browser mode also shows the parent-child relationships between objects but using a browser control.
Information button	Displays basic information about the document's configuration, including which platform and OS versions it supports.
Library button	Displays the Interface Builder Library window. See "The Library Window" (page 35).
Inspector button	Displays the Interface Builder inspector window. See "The Inspector Window" (page 38).
Search field	Lets you search for objects in the document by name. Search results are automatically displayed in the Outline view mode.

In addition to the document toolbar, Interface Builder includes a View menu that you can use to modify the presentation of the document. This menu contains commands that perform the same functions as the toolbar's view mode selector. In addition, you can use this menu to customize the toolbar in various ways.

The content area of the document displays the objects that are created or referenced when the document is loaded by your application at runtime. There are two basic types of objects that can appear in this area: interface objects and placeholder objects. **Interface objects** are the objects that are actually created when the document is loaded and typically comprise the bulk of the objects. **Placeholder objects** are used to refer to objects that live outside of the document but which are intimately tied to the contents of the document.

When you save an Interface Builder document, you save it using either the xib file or nib file format. Both formats store the same information but do so in different ways. **Xib files** are intermediate XML–based files that are intended for use only during the development of your project. Because they are text-based, you can save them in your source-code management system and perform diffs on them. During deployment, xib files are converted to **nib files**, which contain a binary version of the document data and are what your application actually loads at runtime. For any new projects, you should save your documents as xib files. For existing projects, you can also save your documents directly to the nib file format.

Note: Interface Builder documents are often referred to as "nib files" as a convenience and the two terms are considered synonymous. A nib file is simply the end result of an Interface Builder document.

For more information about the objects you put in your nib files, including interface and placeholder objects, see "Nib Objects" (page 55). For more information about the xib and nib file formats and when you should use each one, see "About Nib and Xib Files" (page 47).

Workflow Tools

Interface Builder provides several tools to help you create your nib files. The following sections provide an overview of these tools and how you use them.

The Library Window

The Library window contains the objects and resources you add to your Interface Builder documents. The Library window has three modes that distinguish the type of items you can access. When object mode is selected, the window displays the objects you use to build the user interface of your application; this mode includes windows, menus, views, controls, and many other types of objects. In classes mode, the window displays the classes you can use to build the user interface in your project, including your custom classes. In media mode, the window displays the image and sound resources you can refer to from your objects.

Figure 2-2 shows the Library window in object mode. In all three modes, the window is organized into three primary panes. The Organization pane lists the groups (some of which are organized hierarchically) containing individual items. The Item pane displays the items in the currently selected groups. The Detail pane shows information about the currently selected item.

Figure 2-2 The Library window

The Library window is context sensitive, always showing you the objects supported by the currently active Interface Builder document. You can organize the contents of the Library window in several different ways, including the following:

- Type a string in the Filter field to display only those items whose name or type matches the specified text.
- Select one or more groups in the Organization pane to display items from only those groups.
- Create custom groups and add specific items to those groups.
- Create smart groups to display items that match dynamically-determined criteria, such as when they were last used.
- Rearrange items within a custom group.

For more information on how to customize the Library window, including its contents, see "Customizing the Library Window" (page 159).

Objects and Media

To add objects or media to your document, simply drag the desired items out of the Library window's Item pane and drop them where you want them in your document. Some items, such as windows and menus, can be added only at the top-level of your document. Other items can be dropped onto other items. For example, if you already have a window object in your document, you can drop views and controls directly onto that window (as opposed to on your document). Interface Builder provides feedback about valid drop destinations to help you determine where to place items.

Classes

The Library window provides a central place to view and manage classes. Experienced users of Interface Builder may recognize some of the same features found in the class hierarchy browser in Interface Builder 2.x.

You can use the classes tab to:

- Instantiate a class
- View the lineage of a class
- Find out where a class is defined
- Open a class definition file in Xcode
- View the actions and outlets for a class
- Define custom classes for a project
- Add actions and outlets to a custom class
- Write out updated class files

The classes tab in the Library window contains a hierarchical view of all the classes that Interface Builder knows about for a given Xcode project. This class information is drawn from four sources: project headers, Interface Builder plugins, linked frameworks, and the Interface Builder document itself. These four different sources are reflected in the definitions detail view.

When you select a class, the details pane displays information about the selected class such as its inheritance (lineage detail view), the sources from which Interface Builder has gathered information about the class (definitions detail view), class outlets (outlets detail view), and class actions (actions detail view).

A useful by-product of this central repository of class information is that you can easily instantiate classes that are not defined in an Interface Builder plugin. When Interface Builder builds the list of classes, it looks for the actual objects in the Objects library that most closely match the classes. If you create a subclass of `NSButton` in your code and call it `MyButton`, for example, and you drag `MyButton` out of the Classes library, you drag out an `NSButton`, not just an `NSObject` instance.

Even though Interface Builder has a central place for class management, the definition of your custom classes (changing the superclass, adding outlets and actions, and so on) should be done in Xcode. The ability to add new classes by subclassing in the classes tab or adding actions and outlets is more a prototyping benefit.

Because the classes tab makes it easy to instantiate your custom classes, the recommended workflow becomes:

1. Create your custom classes in Xcode.
2. Instantiate them in Interface Builder by dragging them out of the Classes library.

This eliminates the need to drag out a generic view or object and then set its custom class.

The Inspector Window

The inspector window displays the attributes for the currently selected objects along with controls for adjusting those attributes. Because most objects have a large number of configurable attributes, the inspector window is divided into multiple panes, each of which displays a specific set of attributes. Each pane is then further divided into sections, which can be collapsed to save space.

Figure 2-3 shows the Attributes pane of the inspector window for a Cocoa button. Clicking one of the mode icons along the top of the window selects that pane and displays the associated attribute sections. Changes made in the inspector window take effect immediately for the selected object.

Figure 2-3 The inspector window

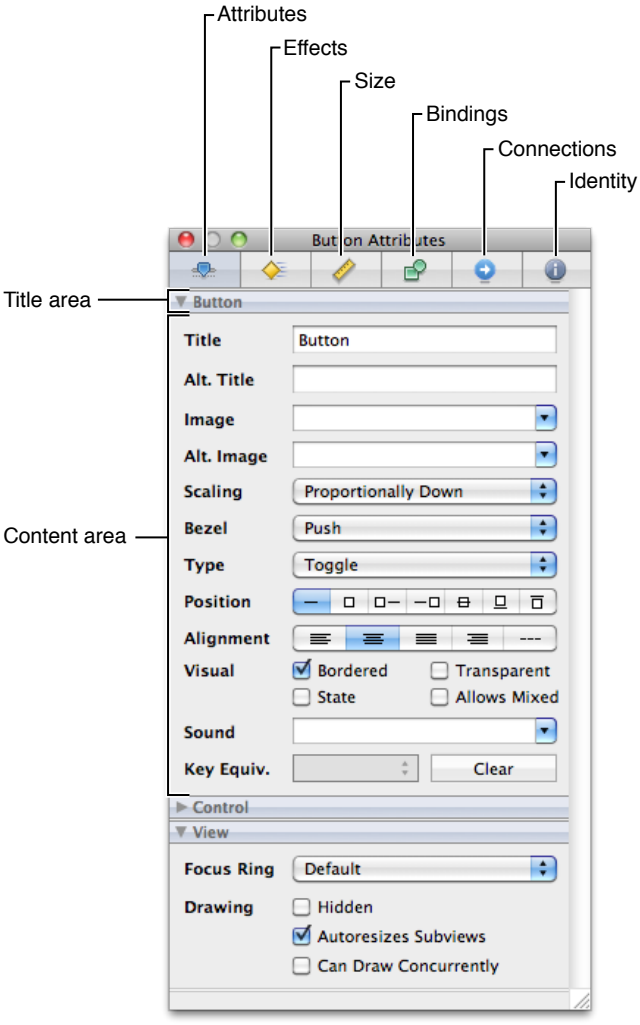


Table 2-2 lists the different panes found in the standard inspector windows and describes their behavior. Cocoa inspectors use all of the panes listed in this table. Cocoa Touch and Carbon inspectors use a subset of panes, which are indicated in the table. This table lists the panes in left-to-right order as they appear in the inspector window.

Table 2-2 Inspector pane behavior

Inspector pane	Description
Attributes	Displays the object-specific configuration attributes. For Cocoa and Cocoa Touch objects, the sections in this pane reflect the classes in the inheritance hierarchy of the selected object. For Carbon objects, the sections reflect opaque type groupings. For more information, see "Object Attributes" (page 105).
Effects	Displays the Core Animation–based and Core Graphics–based attributes associated with the object. You can use this pane to add transparency and shadow effects and apply other types of effects and filters. This pane applies to Cocoa objects only. For more information, see "Attaching Graphical Effects in Mac OS X" (page 106).
Size	Displays information about the size and position of the object and also displays alignment controls and autosizing attributes. This pane applies only to objects that have this information, typically visual objects such as windows, views, and cells. For more information, see "Interface Layout" (page 89).
Bindings	Displays the available bindings for an object and provides an interface for binding those objects to one or more controllers. This pane applies to Cocoa objects only. For more information, see "Connections and Bindings" (page 119).
Connections	Displays the outlets and actions of the object along with information about which ones are currently connected to other objects. In addition to viewing these connections, you can also use this pane to remove a connection. This pane applies to Cocoa and Cocoa Touch objects only. For more information, see "Connections and Bindings" (page 119).
Identity	Displays information that helps identify the object, either at design time or runtime. You use this pane to set the exact type of an object and assign other descriptive information. This information is usually set once and never changed. For more information, see "Object Attributes" (page 105).

Each pane of the inspector window displays as much information as possible when multiple objects are selected. The Attributes pane in particular displays all of the sections that are common to the currently selected objects. For example, if a Cocoa text field and button are selected, the inspector window displays the Control and View sections but does not display the Button or Text Field sections. If a given pane cannot display any relevant information, it instead displays a message to that effect.

The Connections Panel

For documents used on the Cocoa and Cocoa Touch platforms, the **connections panel** is a quick way to create and manage the connections for a particular object. (You can also use the Connections pane of the inspector window.) You display the connections panel by Control-clicking (or right-clicking) an object. The panel is organized into several sections that identify the different types of connections that can be made to and from the object. You use it to create connections for outlets and actions only; to configure Cocoa bindings, you must use the inspector window.

Figure 2-4 shows the connections panel for a button and Table 2-3 (page 40) lists the sections that may be displayed by the panel. If an outlet or action is currently connected, information about that connection is displayed to the right of the outlet or action name. You can create new connections by clicking in the circle to the right of the outlet or action name and dragging to the target object. To break existing connections, click in the break connection box next to the connection target.

Figure 2-4 Connections panel

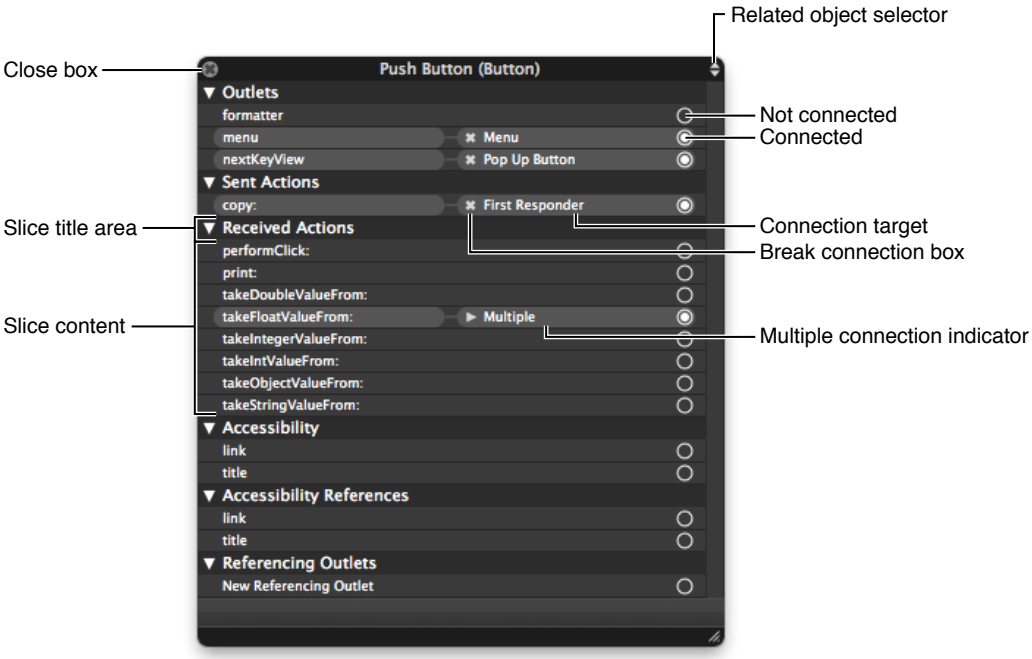


Table 2-3 Connections panel section descriptions

Section	Description
Outlets	Lists the outlets exposed by the object. An **outlet** is a member variable of the class that has the `IBOutlet` keyword associated with it. Outlets let you create inter-object references from within Interface Builder. For information about adding outlets to an object, see "Defining Outlets" (page 142).
Sent Actions	Shows the target of a control's action message. An **action** is a message that is sent by a control in response to a user interaction. For example, when a user clicks a button, the button object sends its action message to the associated target object. In Mac OS X applications, a control sends its action message to a single target object. In iOS applications, a control may send several different action messages in response to different types of user interaction with the control and it may send those messages to multiple target objects.
Received Actions	Lists the incoming actions that the current object is capable of handling. Each of these entries corresponds to a member method of the object. A received action may be associated with multiple objects. For information about defining an object's action methods, see "Defining Action Methods" (page 143).

Section	Description
Accessibility	Lists standard outlets for storing accessibility related information. These outlets are present for all visual elements.
Accessibility References	Lists the source objects that refer directly to the selected object through an accessibility connection.
Referencing Outlets	Lists the source objects that currently refer to this object through an outlet. This section can list multiple sources.

When you control-click an object, the connections panel appears at the point where you clicked. If you click another object without moving the panel, the panel disappears automatically. This is convenient if you want to make a connection quickly and then work on other objects. To keep the panel from disappearing automatically, simply drag the panel away from its original position. Any change in position causes the panel to remain visible. To close a panel that is still visible, click its close box.

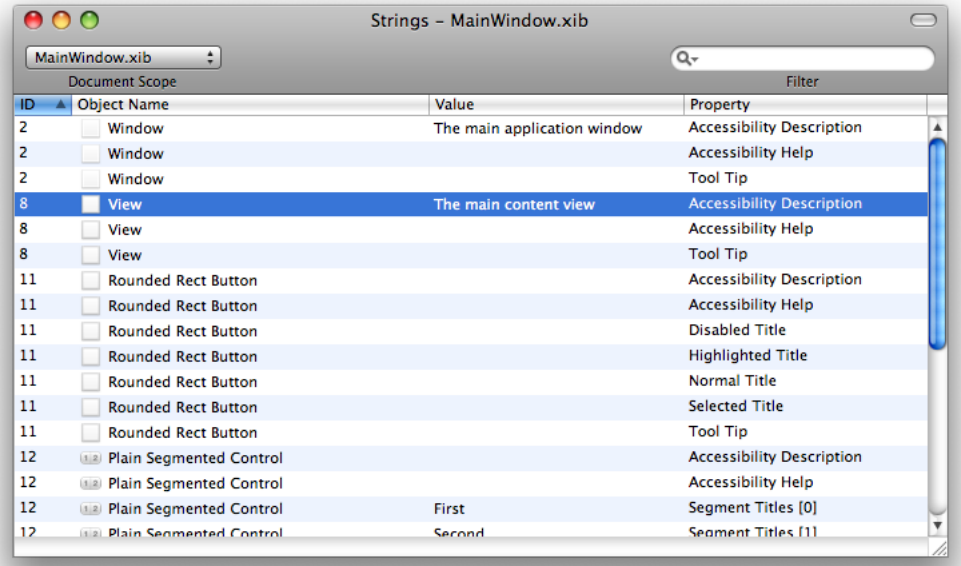
Although the connections panel displays the outlets and actions for the clicked object, you can use the same panel to display the outlets and actions for nearby objects as well. The popup menu in the upper-right corner of the panel lists all of the objects at the current mouse location. You can use this menu to access objects that overlap each other on the design surface and might otherwise be hard to click on directly.

For more information about creating and managing connections in an Interface Builder document, see "Connections and Bindings" (page 119).

The Strings Window

Objects in nib files can have many associated strings. If your application has many nib files, or one large nib file, you may want to ensure that you name similar objects consistently. The easiest way to ensure the consistency of the strings in your nib files is to edit them using the strings window, shown in Figure 2-5. To view this window, choose Tools > Strings.

Figure 2-5 The strings window



The strings window displays the strings from all objects in all open nib files by default. There are a few ways to narrow the list of strings though:

- Use the Document Scope control to show the strings from only the selected nib file.
- Use the Filter control to list only entries that match the specified text.
- Use the Filter control's menu to exclude strings that do not have a value.
- Choose Edit > Find > Find to locate strings whose value matches the specified text.

You can edit strings directly from the Strings window by double-clicking the value of an entry and typing a new value. To sort the strings, click on the appropriate column header.

Interface Builder Behaviors

Interface Builder provides extensive support for undo, drag-and-drop, and pasteboard operations. All actions described in this document are undoable unless otherwise noted. In addition to basic support for these features, Interface Builder also supports several common interaction behaviors. The following sections describe these behaviors.

Modifier Keys

Interface Builder uses the standard modifier keys to modify the way it treats objects. For a summary of modifier key actions, see "Modifier Keys" (page 170).

Selection Behavior and Appearance

Interface Builder helps you select exactly what you want with as few clicks as possible. In most cases, when you click in a window, Interface Builder selects the object directly under the mouse, even if that object is embedded in one or more container views.

In some cases, however, clicking an object might not select that object directly. For example, clicking on a scroller in a scroll view always selects the scroll view first. This happens because the scroll view is considered more important than the scroller. Similarly, when clicking a control, the control, and not its cell, is selected first. The parent object in each case decides whether it wants to handle the initial click. If it does, you can use the **click-wait-click** approach to select the child object:

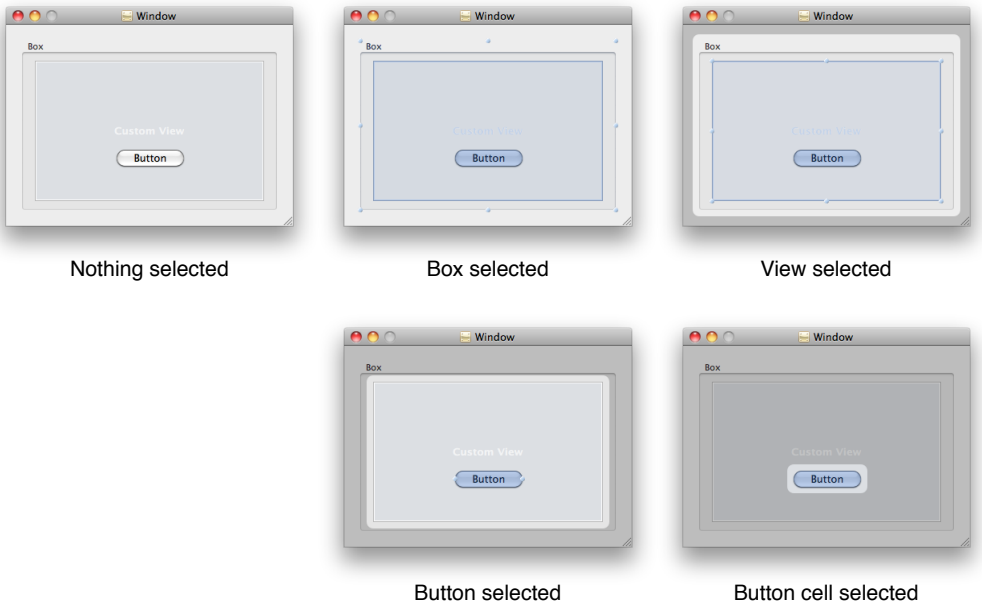
1. Click the parent object to select it.
2. Wait for a second or so.
3. Click the child object to select the child object.

Note: To select objects nested even further down in a view hierarchy, you may have to wait and click multiple times.

Most other selection behaviors work the same way in Interface Builder as they do in other applications. For example, you can use the Shift and Command keys to extend or toggle the selection of items. Interface Builder does impose some restrictions when selecting multiple objects, however. All objects in a selection must have the same parent object.

Figure 2-6 shows how Interface Builder reflects the currently selected item in a window. The selected item is drawn with one or more resize handles and an optional change in shading. The parent of the selected item is drawn normally, but everything outside of that parent's frame is shaded. You can use these visual cues, along with the title of the inspector window, to help determine which item is currently selected.

Figure 2-6 Appearance of selected items



Selecting Hidden Objects in a Nib File

For the most part, you select objects by clicking directly on them directly from the design surface. In some cases, however, it might be difficult to click an object directly. For example, a tab-less tab view acts as a container for multiple subviews but only one subview at a time is visible. What's more, the tab view itself is typically obscured by its own subview. In such a case, you can select any of the obscured views using one of the following techniques:

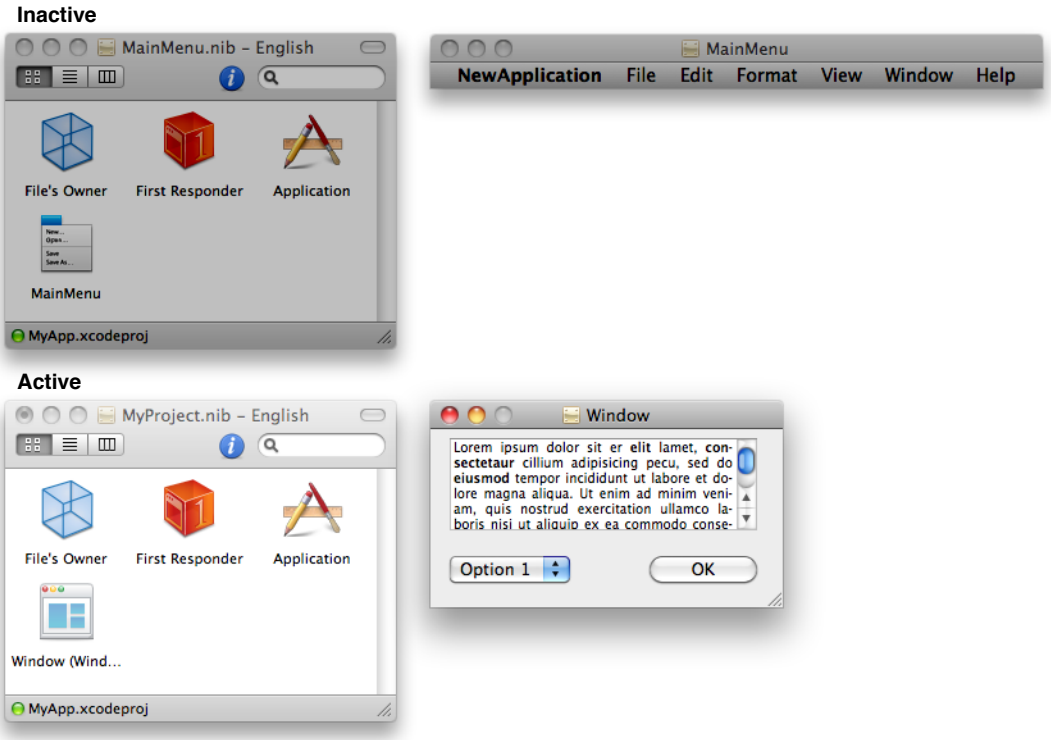
- Set the viewing mode of your Interface Builder document window to Outline or Browser mode and select the object there.
- Control-Shift click (or Shift right-click) the object obscuring the desired view and select the desired object from the menu that appears.

Control-Shift clicking an object is a way to select any of the objects that lie directly beneath the mouse. When you do it, Interface Builder displays a menu, from which you can select the desired object.

Multiple Document Shading

Although you can have multiple Interface Builder documents open at the same time, only one can be active at any given time. To avoid confusion during editing, any windows that belong to an inactive document are dimmed to indicate their relationship to their state (see Figure 2-7). This makes it easier to determine which windows belong to the current document and which document is currently active. Even if a particular document is inactive, you can still drag and drop items between it and other documents, however. Doing so copies the item from the active document to the target document. You can also use the pasteboard to move items.

Figure 2-7 Shading of active and inactive documents



Nib File Management

Nib files play an integral role in the development of your applications. They are a powerful type of resource that you use to store runtime objects and interface-related content. They are also the fundamental document type of the Interface Builder application.

Interface Builder 3.0 and later offer many options for creating and working with nib files themselves. Understanding these options can help save you time and effort during development. This chapter offers guidelines to help you create nib files that integrate well with your Xcode projects.

About Nib and Xib Files

Files in the nib and xib formats play a very important role in the creation of applications. In their primary role, nib files simplify the code you have to write to create your application's user interface. They provide a loadable set of objects that replace the code you would write to create windows, views, and other interface-related items. In some applications, nib files also provide a means to integrate these interface items with the existing objects in your application.

The nib and xib file formats themselves provide you with options for the development of your projects. Although they represent the same information, you use each type of file differently. The xib file format is preferred during development because it provides a SCM-friendly format, and because xib files can be compared with the diff command. At build time, Xcode automatically converts your project's xib files to nib files so that they can be deployed with your application. If you have existing nib files, however, you can also continue saving to that same format.

What Goes in a Nib File?

Although the contents of a nib file can vary greatly in theory, in practice, there are usually a handful of standard configurations that developers include. The reason is that most developers use nib files for a very specific purpose: to load user interface items. This is particularly true for Carbon nib files, which can be used to store only user interface components, such as windows and menus. Nib files for other platforms offer more support for non interface components and for more complex configurations of your objects. For example, when you open a nib file for a Cocoa application, you might see any of the following items at the top level of the nib file:

- Window resources
- Menu resources (both menu bars and individual menus)
- Views
- Formatter objects
- Controller objects (view controllers, array controllers, object controllers, and so on)
- Core Data managed object contexts

- Program-specific custom objects
- Placeholder objects, including File's Owner

Some of these objects must be at the top level of the nib file. For example, windows and menus must always be at the top level of a nib file. In addition, controller objects are almost always at the top level. This is because controller objects cannot be embedded inside windows, views, or menus. (Cells and formatters are special because they work in conjunction with a view or control to implement its appearance.) Most other view-based objects are typically situated inside a window or view, although they can appear at the top level of the nib file as well.

Nib File Design Guidelines

The following guidelines can help you create nib files that achieve your application design goals quickly and efficiently.

Keep Your Nib Files Small

Most new projects in Xcode come with one or two preconfigured nib files. A mistake made by many developers who are new to Interface Builder is to place all of their application's windows and menus in these one or two nib files. The reason for the mistake is often convenience. (The template project usually loads the preconfigured nib files automatically, which saves the developer from having to add more nib-loading code.) Unfortunately, relying on this convenience can often lead to diminished performance and added memory pressure on your application.

When a nib file is loaded into memory, it is an all-or-nothing endeavor. The nib-loading code has no way of knowing how many objects are in the file or which ones are important, so the entire file must be loaded into memory. From this in-memory data, individual objects are then instantiated. For Mac OS X and iOS applications, all of the objects in the nib file are instantiated right away so that outlet and action connections can be reestablished. If your application uses only some of the nib-file objects initially, having all of the objects in memory is a waste.

For all projects, it is better to design each nib file so that it contains only those objects that are needed immediately in a given situation. When loaded into memory, such a nib file uses the smallest amount of memory possible while still having everything it needs to get the job done. Here are some design choices to consider when organizing your nib files:

- For your application's main nib file, include only your menu bar (or in the case of an iOS application, just the main window).
- For document nib files in Mac OS X, include only the document window and the objects (such as controllers) needed to display that window.
- For other nib files, focus the nib file on a key object, such as a single window or panel that you intend to display. All other objects in the nib file should then facilitate the immediate operation of that window or panel.
- For windows that change their embedded view hierarchies, if the hierarchy changes infrequently, consider storing any additional hierarchies in separate nib files. Load each view hierarchy only as it is used.

If you already have large nib files, you can use Interface Builder's refactoring tools to break them into several smaller nib files. For information on how to use Interface Builder's refactoring tools, see "Refactoring Your Nib Files" (page 53). For information about how to load nib files explicitly from your code, see *Resource Programming Guide*.

Choose Appropriate Controller Objects

In Cocoa and Cocoa Touch nib files, the File's Owner placeholder object provides the key link between your application and the objects in the nib file. When you load the nib file, you must provide the nib-loading routine with a pointer to the object that should become the File's Owner. As part of the loading process, the nib-loading code automatically recreates any connections between the object you specify and the nib file objects that have connections to the File's Owner.

As you design the architecture of your application, it is important to consider which objects you want to manage your nib files. The presence of only one File's Owner placeholder object is not without good reason. It is usually best to have a single object coordinate the loading and management of a nib file and its contents. This single point of contact provides the desired barrier between your application's data model and the visual elements used to present that data model and is at the heart of model-view-controller design.

Beyond the File's Owner object, you can create additional controller objects directly in your nib file to manage subsets of the nib file. Using multiple controllers in this way lets you compartmentalize the window's behavior into more manageable chunks. For example, if your window has multiple panes of disparate information, you could create separate controller objects to manage each pane. Each controller would continue to go through the File's Owner to obtain additional information.

In iOS applications, it is also possible to include placeholder objects besides File's Owner in your nib file. These additional placeholder objects are almost always used to represent navigation controllers and other view controllers already in use by your application. The presence of these additional placeholder objects does not diminish the role of File's Owner though. The File's Owner object is still responsible for coordinating the overall behavior of the nib file's contents.

Place Appropriate Objects at the Top Level of a Nib File

The objects at the top level of a nib file represent the main objects of interest to your application. Although technically you can place almost any object at the top level of a nib file, in most cases, doing so is rarely practical or necessary. Most applications use nib files to load a particular piece of the user interface, usually a window or menu. For Carbon applications, windows and menus are the only thing you can include at the top level of a nib file.

In addition to windows and menus, nib files for Cocoa and Cocoa Touch applications can include controller objects and formatters at the top level. Creating these objects from the nib file is often more convenient than creating them programmatically. And if their entire purpose is to manage the objects inside the nib file, it is also more practical to include them as part of the nib file.

Choosing the Best File Format

The nib file format is the original file format supported by early versions of Interface Builder. The nib format is still used today as the deployment format for object resources you load into your application at runtime. In Interface Builder 3.0, the nib format was expanded to include some additional design-time information. This updated version of the nib format is still backwards compatible with the nib-loading infrastructure but is not compatible with older versions of Interface Builder.

The xib file format was introduced in Interface Builder 3.0 as a development-time format and was conceived as a way to provide tighter integration with your Xcode projects, particularly in the areas of SCM support, diff support, and refactoring. Xcode automatically converts files in the xib format to the nib format at build time.

Because there are two supported file formats for Interface Builder documents, you have to choose which format to use for your own projects. If you're developing for iOS, the choice is easy—you must use the xib file format for your project. For Mac OS X, you may use either the xib or nib file format. In almost every case, you should use the xib format. The nib format is recommended for a project only if the project target is an Interface Builder plug-in that depends on itself.

Interface Builder 3.0 and later fully supports opening nib files created with Interface Builder 2.5.x and earlier. Nib files created or modified with Interface Builder 3.2 and later can't be opened with Interface Builder 2.5.x and earlier.

Nib Files and Localization

Because nib files can store user-visible strings, their contents must be localized along with the rest of your application during the localization process. Because nib files are external to your code, however, the localization process is straightforward. Like other resource types, you can store localized versions of your nib files in language-specific project directories of your application bundle.

For more information about the localization process for nib files, see "Localization" (page 153).

Creating a Nib File

To create a nib file in Interface builder, choose File > New. Interface Builder displays the template chooser for creating your nib file. From this panel, you select the target platform and the starting template you want to use.

The starting templates simplify the nib creation process by providing you with preconfigured nib files that you would use in typical situations. The templates also provide you with objects that are configured correctly for the target platform. Each application platform has subtle differences, both in what the nib file can contain and in the supported frameworks. For example, the Cocoa Touch platform supports classes from the UIKit framework while the Cocoa platform supports classes from the AppKit framework. The platform you choose therefore determines what content shows up in the library and inspectors.

The template chooser also includes nib templates for use with the IB SDK. These nib files are Cocoa nib files that you use to create Interface Builder plug-ins, which are add-on modules used to extend the set of objects in the library. For more information on how to create plug-ins, see *Interface Builder Plug-In Programming Guide*.

Note: For the Carbon and Cocoa platforms, the only way to include a menu bar resource in your nib file is to choose the appropriate template. The Application and Main Menu templates typically contain menu bar resources while other templates do not. Because menu bar resources are not included in the library, starting with these templates is your only option for creating one.

After creating a new nib file, the next step is to add your custom objects to it. For information about the objects you can include in nib files, see "Nib Objects" (page 55).

Saving Nib Files

When you are ready to save your Interface Builder document, choose File > Save from the menu. If you are working on a version 2.x nib file, Interface Builder transparently upgrades it to version 3.x. For the iOS platform, you must save your nib file using the version 3.x xib file format (`.xib` extension). For the Mac OS X platform, you can save your document using one of two file formats:

- Version 3.x development-time format (`.xib` extension)
- Version 3.x deployment format (`.nib` extension)

Depending on the situation and your needs, you may find yourself using one or both of these formats for a given project. For guidance on which format to choose for your project, see "Choosing the Best File Format" (page 49) and the following sections.

Saving in the Xib File Format

The development-time xib file format is a text-based flat-file format introduced in Interface Builder 3.0. This format was added to make it easier to check Interface Builder documents in and out of source-control systems. Files of this type have a `.xib` extension and can be used only within the Xcode and Interface Builder environments. You cannot load `.xib` files at runtime from your application code. Instead, when you build your Xcode project, Xcode compiles this format down into a deployable nib file and adds that file to your project bundle.

The xib file format is the default format for Interface Builder. You should use this format for any new (or existing) projects you are developing using version 3.0 or later of Xcode and Interface Builder. To save a file using the xib format, do the following:

1. Choose File > Save (or File > Save As) to display the Save panel.
2. Select the XIB 3.x format from the File Type pop-up menu.
3. Click Save.

Important: Although it is a text-based format, you should never edit a xib file by hand. The xib file format represents a flattened archive of the nib file object graph. Even if you are cautious, making changes to one portion of this graph could cause unexpected changes to other portions. You should also avoid using source code management tools to merge a xib file with another file. A textual merge, even one without conflicts, will likely produce an invalid xib file.

Saving in the Nib File Format

The deployment nib file format provides compatibility with, and facilitates the direct modification of, existing nib files. You might use this format when modifying a nib file for an existing project or when creating nib files for an Interface Builder plug-in that depends on itself.

Interface Builder supports version 3.x of the deployment format. The 3.x version is a more modern format supported by Interface Builder 3.0 and later. This format supports new objects that are available only in Interface Builder 3.0 and later.

Important: If you are developing applications using Xcode 3.0 or later, you should avoid using the deployment nib file format during development. Instead, you should use the development-time nib file format and let Xcode create the appropriate deployment nib files for you. The development-time format makes it easier to integrate your nib file code with source-control systems and build scripts.

To save a file using the deployment nib file format, do the following:

1. Choose File > Save (or File > Save As) to display the Save panel.
2. Select the NIB 3.x format from the File Type pop-up menu.
3. Click Save.

Using Image and Sound Resources in a Nib File

To view the image and sound resources contained in your Xcode project, use the media tab of the Library window. This tab displays your Xcode project's custom resources and some of the common resources available in the system. The contents of this tab are essentially read-only and intended as a way to browse your project resources without having to go back to Xcode.

Views that support image and sound resources usually provide a configurable attribute in their inspector window for selecting the appropriate file from the media browser. You can also integrate image and sound resources directly into your nib file using the following techniques.

- Drag an image from the media browser and drop it onto a window to create an image view configured with that image.
- Drag an image from the media browser and drop it onto a toolbar, button bar, or tab bar (depending on the platform) to create a new item for that view.
- Drag an image from the media browser and drop it onto a control to assign that image to that control's cell (where applicable).

Although you can reference image and sound resources from your nib file, those resources are still stored outside of the nib file itself. Interface Builder gets the list of available resources from the Xcode project associated with your nib file. Therefore, if you want to use image and sound resources in your nib files, add them to your Xcode project.

In addition to your project resources, the Cocoa and Cocoa Touch platforms make some standard system images available for applications to use. These images are displayed in the Media browser by default but you can also refer to these images by name. The reference documentation for the associated platform lists the constants containing the name strings you use to access them. In nearly all cases, the names used by Interface Builder are a shortened version of the constant name. For example, in Cocoa applications, Interface Builder uses the string "NSBonjour" to represent the image identified by the `NSImageNameBonjour` constant. For a list of constants you can use in Cocoa applications, see *NSImage Class Reference*. For Cocoa Touch applications, these constants are spread across several classes in *UIKit Framework Reference*.

When in doubt about whether a control supports image or sound resources, check the inspector window. If a control includes an image or sound field as one of its attributes, then you can assign the appropriate resource to it. (For example, you can assign both an image and sound resource to a push button control in Cocoa.) In addition to typing the resource name in the appropriate field, the inspector window typically includes a drop-down list that includes the known resources from your Xcode project.

Refactoring Your Nib Files

It is a common mistake for developers to use their application's main nib file to store several unrelated windows and controller objects. A better design pattern is to use several smaller nib files, each containing only the objects and controllers needed to implement a single window or interface. Smaller nib files promote better efficiency and a smaller memory footprint for your application. If your application already has large nib files, however, breaking up those files by hand is difficult, tedious, and error-prone. To help with the process, Interface Builder provides a refactoring tool to do the job for you.

To use Interface Builder's refactoring tool, open the nib file you want to refactor and choose File > Decompose Interface. Interface Builder iterates through the file's contents to build a map of the object relationships in the nib file. The tool looks at connected outlets, actions, bindings, and at the window and view hierarchies. From that map, it then identifies distinct, unrelated groups of objects and creates new nib files for each distinct group. The command does not change the contents of your original nib file.

Note: Interface Builder's refactoring support is also used by Xcode's refactoring engine to propagate changes to your nib files during a refactoring session. For more information on refactoring your Xcode programs, see "Refactoring Code".

Converting iOS Nib Files

If you want to adapt an existing iPhone user interface for iPad, you can use one of Interface Builder's Create commands to create a new iPad document. Conversely, to adapt an existing iPad interface for iPhone and iPod touch, you can use a Create command to create a new iPhone document. The Create commands are intended to help you get started designing a new interface; they do not exactly replicate the existing interface. After creating an iPhone version of an iPad user interface, for example, you still need to modify the new interface for the target device.

To use this feature, open the nib file you want to convert and choose one of the two Create commands in the File menu. Interface Builder creates an untitled document according to the command you choose. If you choose File > Create iPad Version, the new document contains an iPhone/iPod touch interface without autosizing the views. If you choose File > Create iPad Version Using Autosizing Masks, Interface Builder respects the current autosizing settings in the views being converted. Two complementary Create commands exist for iPad documents.

Nib Objects

The editing environment of Interface Builder is organized around the direct manipulation of objects. How you manipulate a given object, though, depends on the type of the object and its typical relationship to other objects.

Views are the most common type of object used in Interface Builder. Views are rectangular regions that provide some visual content and optional event-handling behavior. Controls are a special type of view that respond to user interactions and send runtime messages to an interested target object when those interactions occur.

Windows and menus are also the main building blocks of many Interface Builder nib files. Most nib files are created around a specific window, menu, or view, and all of the other objects in that nib file are there to support that window, menu, or view.

In addition to visual objects such as windows, views, and menus, nib files used with Mac OS X and iOS applications can also include non-visual objects. Because Mac OS X and iOS applications use the Model-View-Controller design paradigm, most non-visual objects are controller objects whose job is to manage all or part of the user interface stored in the nib file.

The following sections introduce you to the key objects you are going to find in a nib file and how you use them to build your interface. These sections are not intended as a comprehensive catalog of all the objects you can add to a nib file. Instead, these sections group together objects with similar behavior and treat them as one in order to simplify the explanations of how they work.

iOS Interface Objects

This section describes the objects typically available on the iOS platform. Many objects available on the Mac OS X platform are also available in iOS, although in several cases the way you use those objects is slightly different. In addition, iOS introduces several new types of objects that have special behaviors and usage patterns. The following sections provide you with information about the objects available on the iOS platform and how you use them.

Windows

A typical iOS application has only one window, which provides the backdrop for all of the application's content. Applications can include additional windows, if desired, but doing so is rare and generally not recommended.

Adding a Window to a Nib File

Because it is an integral part of the application's interface, you usually put your application's main window in the main nib file so that it is loaded at launch time. The main nib file typically comes preconfigured with a window object that you can simply modify. You can also use the Application and Window nib file templates to create a new nib file with a window. To add a window to your nib file manually, drag a window object from the library and either drop it onto the desktop or onto your Interface Builder document.

Dropping a window object onto the desktop adds that window to the top level of the active Interface Builder document window. In iOS, windows do not have title bars or other visual elements that would require different window styles and Interface Builder automatically sets the size of each window to the size of the device's screen. In addition, if your window is configured to display a status bar, Interface Builder displays a mock status bar to help you position your content. This status bar is not saved with your nib file.

When you double-click a window object, Interface Builder opens that window on the desktop (or makes it the active window if it is already open). The content area of the window represents the **design surface** and is where you build the appearance of your window.

Removing a Window from a Nib File

To remove a window from your nib file:

1. Select the window in your Interface Builder document window.
2. Press the Delete key (or choose Edit > Delete from the menu).

Removing a window removes the window, its contents, and any connections to other objects from the nib file.

Views and Controls

In iOS, everything that appears onscreen descends from the `UIView` class. In practical terms, views and controls are rectangular regions that display data and are capable of handling events. Buttons, text fields, image views, and table views are all examples of views, with buttons being a specific type of view called a control.

It's unusual to add views to an iOS application's main window, set up in the `MainWindow` nib file. Instead, iOS applications generally contain one or more additional nib files (where the File's Owner is a view controller), each with a view at the top level of the file. In the top-level view, you place other subviews (buttons, text fields and so on). Applications use these additional nib files to display different screens' worth of content in the main window. To learn more about using these files, see "View Controllers" (page 61).

Adding Views to a Nib File

To add a subview to a nib file, you do the following:

1. Open the nib file's top-level view so that its design surface is visible.
2. Open the library and locate the desired view or control. (Views and controls are grouped into categories to help you find them. You can also type the control name in the Filter box to search for items.)

3. Drag the view or control out of the Library window.
4. Drop it onto the design surface.

As you drag a view onto the design surface, Interface Builder highlights different portions of the surface to show you where the drop would occur. Upon dropping the view, the highlighted object becomes the parent of the dropped view.

Removing Views from a Nib File

To remove a view from the top level of your nib file:

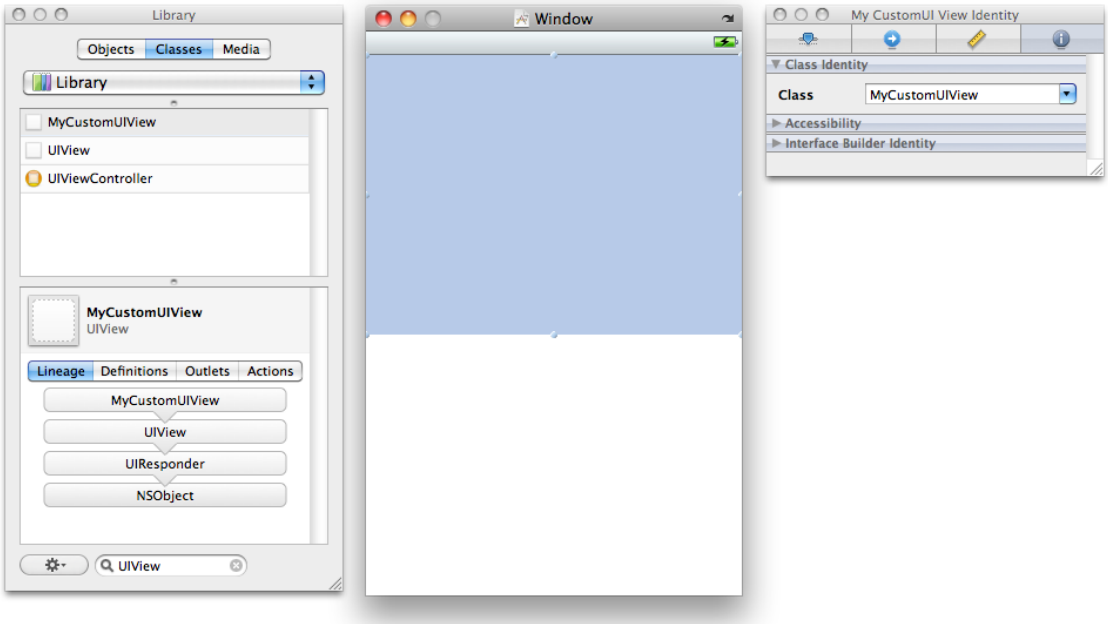
1. Select the view.
2. Press the Delete key (or choose Edit > Delete from the menu).

Removing a view removes its subviews and any connections and bindings to other objects from the nib file.

Custom Views

Although the standard system views and controls provide a variety of choices for building an interface, there are many times when you may want to provide customized behavior for your application. If your nib document is associated with a Cocoa Touch project, Interface Builder is aware of any custom views declared in your source code. You can find your custom views listed in the Classes tab of the Library window. Dragging a custom view object to a window (or to the top level of your nib document) creates an unadorned rectangular area representing the space occupied by the object. Interface Builder sets the object's class name in the identity inspector, as illustrated in Figure 4-1. To learn about setting the class identity of a generic view object, see "Setting the Class of an Object" (page 139).

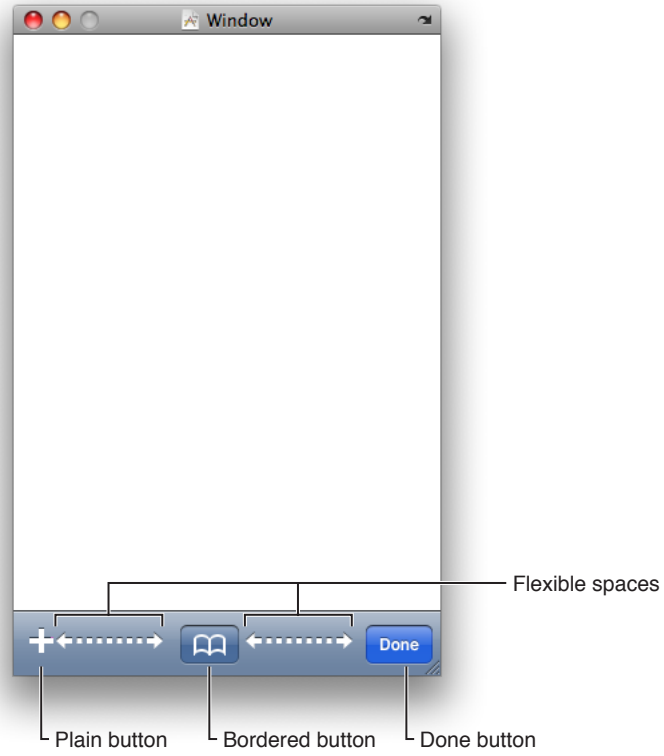
Figure 4-1 Custom UIView object



Toolbars

Toolbars contain a collection of buttons representing frequently used commands in an application. Toolbars are typically located along the bottom edge of your user interface. You can configure the appearance of a toolbar in different styles to suit your interface needs. You can also configure the buttons in a toolbar using several different styles, which are shown in Figure 4-2.

Figure 4-2 Bar button items in a toolbar



To add a toolbar to your user interface, drag a tool bar view from the library and drop it in your window or view. The default toolbar contains a single bar button item, but you can add additional items by dragging them from the library. You can configure the properties of each bar button item using the inspector. You can also edit the title of a bar button item by double-clicking it on the design surface.

Setting the Appearance of Bar Button Items

Bar button items you drag from the library are configured as custom buttons by default. You can specify a title for the button or specify a custom image you want to use in the button instead. Before specifying an image for your bar button item, you must add it to the associated Xcode project. Interface Builder provides a list of the available images in the combo box associated with the Image field of the inspector.

In addition to custom buttons, you can configure bar button items using several standard images and titles. When a bar button item is selected, the Identifier field of the Attributes inspector lists the type of the button. Choosing a type other than Custom lets you create buttons representing standard system actions.

Positioning Buttons on the Toolbar

As you add new buttons to a toolbar, Interface Builder places them at the left edge of the toolbar by default. If you want to position buttons along the right edge of the toolbar, or space them out along the entire toolbar length, you must use a specially configured button to insert the appropriate amount of space. You can configure buttons to occupy a flexible or fixed amount of space.

Flexible-space buttons expand to fill the available space on the toolbar, pushing other buttons all the way to the right edge of the toolbar. Fixed-space buttons fit the available space as best they can initially and can be reconfigured later to specify the desired amount of space you want. To change the size of a fixed-space item, open the Size inspector and change the value in the Width field located in the Bar Button Item Size section. Flexible-space buttons ignore the value in this field.

Note: If your application supports both landscape and portrait orientations, you should favor flexible-space items over fixed-space items whenever possible. When changing orientations, flexible-space items expand or contract as needed to fill the available space.

Configuring the Action of a Bar Button Item

When the user taps a bar button item in your toolbar, the item delivers an action message to a designated target object. The way in which you configure the action message of a bar button item is different from how you configure the events for other iOS controls. (In fact, the process is more like the target-action connections created between controls and objects in Mac OS X.) A bar button item sends its action message to a single target object only and sends it only when the user's finger lifts from the button surface. The handler method that your target object uses to receive the message must have the following format:

```
- (IBAction)myActionHandler:(id)sender;
```

You connect the bar button item to its target object using either the Connections inspector or the connections panel. The bar button item has a single `selector` action that you can connect to the action method of the designated target object. You can initiate the connection either from the bar button item or the target object. When initiating a connection from the bar button item, you must start from the selector action and release the mouse button over the target object, selecting the desired method of that object to complete the connection.

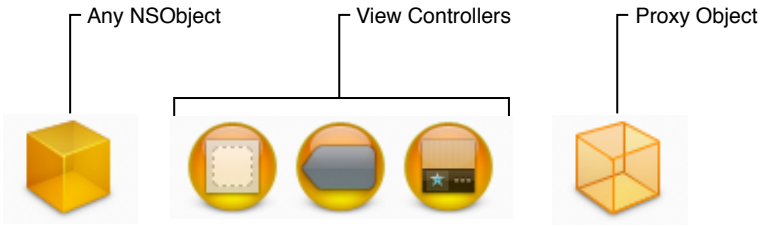
For more information about creating connections between objects, see "Connections and Bindings" (page 119).

Controller Objects

In addition to visual objects, Cocoa Touch nib files can include any type of custom object (including non-visual objects) needed by an application. The ability to include any instance of `NSObject` is typically used as a way to facilitate the Model-View-Controller design pattern used by iOS applications. The non-visual objects in a nib file act as controllers for the visual objects.

Figure 4-3 shows the basic object types you can add to a Cocoa Touch nib file. In addition to the generic object type, Cocoa Touch nib files can include two other special object types. View controller objects provide custom management for an application's user interface and are used heavily in iOS applications. Because an application's interface may be spread over multiple nib files, Interface Builder also includes a placeholder object, which you can use to represent any additional objects that are distinct from the File's Owner placeholder.

Figure 4-3 Controllers and custom objects in Cocoa Touch nib files



View Controllers

Because an iOS application has only one window, the application changes the window's content by changing the content views displayed in the window. The **content view** is the portion of an iOS window that you use to display your application's custom content. The content view is not necessarily a single view but may in fact comprise several views embedded inside a parent (or root) view.

Management of the content view is the responsibility of a view controller object, which coordinates the movement of that content on and off the screen. Because it is a controller, you can also add custom logic to your view controller classes to manage interactions between the content view and your application's data model.

In addition to providing navigation and tab bar controller objects, Interface Builder provides a generic view controller object that you can use for managing individual views. The most common way to use a view controller is as the File's Owner of a nib file, but you can also instantiate them inside your nib file as needed.

Using a View Controller as File's Owner of a Nib File

To use a view controller as the File's Owner of your nib file, do the following:

1. Create a new nib file using the Cocoa Touch View template and configure it as follows:
    a. Set the class of the File's Owner placeholder object to the class of your custom view controller.
    b. Connect the view outlet of the File's Owner to the view provided by the template.
2. Open the view and add any additional subviews to it to define your user interface.
3. Connect any additional outlets and actions.
4. Save the new nib file and add it to your Xcode project.

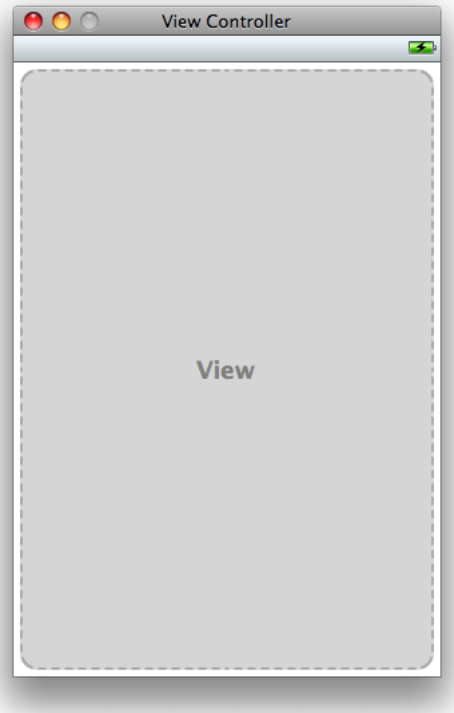
Nib files configured in this way are associated with the corresponding view controller class at runtime. When you create a new instance of your view controller, you pass the name of your nib file to the `initWithNibName:bundle:` method of the `UIViewController` class. The first time your application asks the view controller for its view, the view controller loads the associated nib file and returns the view attached to its `view` outlet. The view controller continues to manage the contents of the nib file internally, purging the views as needed during low-memory conditions and reloading them later as needed.

Nib files configured in this manner are often used to implement tab bar and navigation-style interfaces. The content view for each distinct screen is stored in its own nib file and associated with a view controller class. The navigation and tab bar controllers then manage the loading and unloading of each view controller and its associated nib file.

Adding a View Controller to Your Nib File

If you add a view controller to the top level of your nib file, double-clicking that controller object opens the editor window shown in Figure 4-4. This window acts as the design surface for your view controller's view. Views you add to it are embedded as child objects of the view controller object in your Interface Builder document. The root view of your hierarchy is also assigned to the `view` outlet of the view controller by default.

Figure 4-4 View controller editor window



The view controller editor displays a status bar (and other top bars and bottom bars) if the properties of that view controller indicate that one is present. These elements are not saved with your nib file and are provided to help you position your views and other content. You should configure the attributes of your view controller to match the presence of the status bar or other elements in your application window.

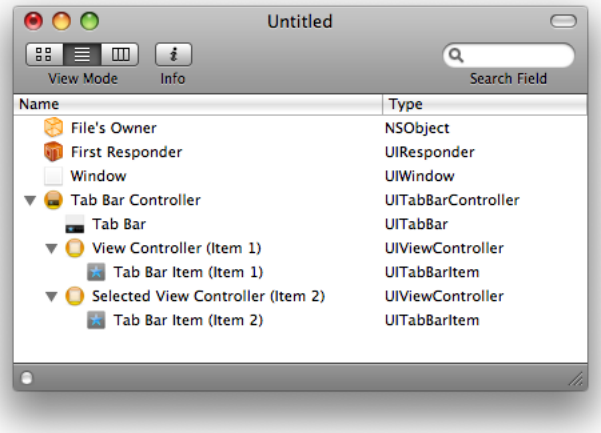
Tab Bars

Tab bars give the user a way to switch between different user interface modes in an application. Tab bars are typically used by applications to manage large amounts of information or to present information in different ways for the user. For example, the Clock application uses a tab bar to change between different clock-related utilities, as shown in Figure 4-5. Pressing each button in the tab bar displays a unique interface for that mode.

Figure 4-5 Interface modes of the Clock application

In Interface Builder, the way to configure a tab bar interface is not by adding views to your window. Instead, you need to add a tab bar controller object to the top level of your nib file and configure your views using the custom editor for that object. When you add a tab bar controller object to your document, Interface Builder actually adds several other objects. In addition to the tab bar controller, it adds a tab bar view and two generic view controllers, each of which contains its own tab bar item. Interface Builder groups these objects together under the tab bar controller, as shown in Figure 4-6.

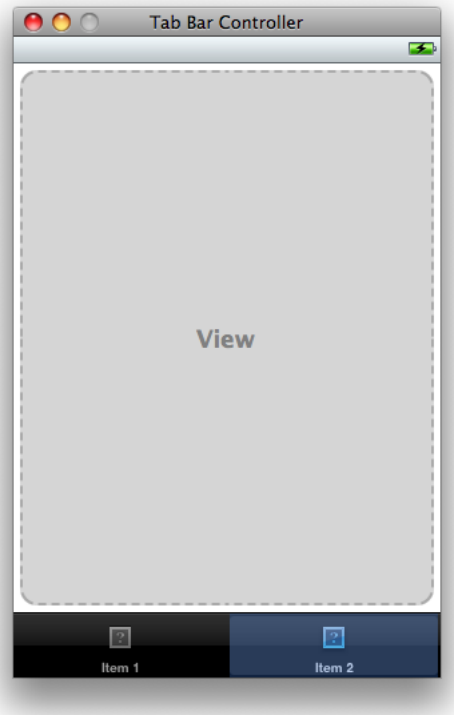
Figure 4-6 A tab bar controller and its default set of objects



The grouping of the tab bar objects as children under a tab bar controller reflects the object graph relationships between those objects. A tab bar controller's children consist of its tab bar view and the view controllers used to manage each of its tabs. The view controllers are actually stored in the `viewControllers` property of the tab bar controller object, which can also be set programmatically. The tab bar view is added to the overall view associated with the tab bar controller, which is stored in the controller's `view` property.

The relationship of objects in the document window also reflects the way those objects are presented by Interface Builder. When you double-click a tab bar controller object, Interface Builder displays the tab bar controller editor window, shown in Figure 4-7. This editor window reflects the child objects of the tab bar controller and is where you configure your tab-based user interface. Items you add using this interface are similarly added as children to the tab bar controller or the view controller that owns them.

Figure 4-7 Tab bar controller editor window



The tab bar controller editor displays a status bar (and other top bars and bottom bars) if the properties of that view controller indicate that one is present. These elements are not saved with your nib file and are provided to help you position your views and other content. You should configure the attributes of your tab bar controller to match the presence of the status bar or other elements in your application window.

Selecting one of the tab bar items in the tab bar controller editor window shows the views associated with the corresponding tab. In fact, clicking a tab bar item once selects the view controller that manages that item. Clicking the same item again selects the actual tab bar item.

Although the tab bar controller lets you edit the objects associated with the tab bar interface, simply editing those objects does not make them appear in your application window. To display your tab bar interface, you must programmatically retrieve the tab bar controller's view and add it to your window as a subview. For example, if your application delegate has outlets referring to your tab bar controller object and application window, its `applicationDidFinishLaunching:` method would need to look something like the following:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    [window addSubview:[myTabBarController view]];
}
```

For information about configuring tab bars and tab bar controllers programmatically in your application, see "Tab Bar Controllers" in *View Controller Programming Guide for iOS*.

Adding and Removing Tabs

There are several ways to add a new tab to your tab bar interface:

- Drag the desired view controller from the library and drop it onto the tab bar in the tab bar editor window.
- Select the tab bar controller object and add a new tab using the plus (+) button in the Attributes inspector.
- Drag a tab bar item from the library and drop it on the tab bar editor window.

Whenever you add a new tab to your interface, Interface Builder adds both a new tab bar item and a view controller to your nib file. When dragging items over the editor window, Interface Builder shows you the proposed placement for the new item. You can rearrange items later by dragging them around the tab bar or by modifying the order of the view controllers in your document window.

If you want an existing tab to use a different view controller class, select the tab bar controller object and open the Attributes inspector. The Tab Bar Controller section contains a table listing the view controller objects associated with the tab bar. To change the class of one of these view controllers, use the provided pop-up menu in the Class column. If you are adding a new tab, as opposed to modifying an existing tab, drag the desired controller object from the Library window and drop it directly onto the tab bar in the tab bar editor window.

To remove a tab from your interface, select the corresponding tab bar item and choose Edit > Delete or simply press the Delete key.

Configuring the Tab Bar Items

Each tab bar item can be configured with a name, an image, and a badge value. You can configure all of these properties using the Attributes inspector. Before setting the image for a tab bar item, you must first add that image to the associated Xcode project. (Interface Builder provides a list of the available images in the combo box associated with the Image field of the inspector.) You can also double-click the title string of a tab bar item to edit the title in place.

In addition to specifying custom strings and images for your tab bar items, Interface Builder provides a list of standard tab bar items that you can use in your applications. To use one of these items, select the appropriate value from the Identifier menu. It is important to remember that these standard items determine only the appearance of the tab bar item. Your application is still responsible for providing the associated content views and behavior of all items on the tab bar.

Because of the way tab bar items are organized, clicking an item in the editor window selects the view controller that owns that item. Clicking that item again selects the tab bar item associated with that view controller. You can select either the tab bar item or the view controller directly from your Interface Builder document window while it is in the outline or browser view mode.

Note: If your tab bar contains more than five items, do not create a More button to allow the user to configure which items to display in the tab bar. Instead, configure the `customizableViewControllers` property of the tab bar controller programmatically as described in "Tab Bar Controllers" in *View Controller Programming Guide for iOS*. Configuring this property automatically adds a More button with the desired behavior.

Configuring the Views for a Tab

Each tab bar item has an associated view controller to manage the set of views displayed while in the given mode. There are two ways to configure the views associated with a single tab:

- Use one or more separate nib files to specify the views. (Recommended)
- Embed the views in the same nib file that contains the view controller.

By their nature, tab bar interfaces add complexity (and many more views) to an application's user interface. Although you can embed all of your application's views in the same nib file as your tab bar controller object, doing so is not terribly efficient. Storing all of those views in the same nib file causes them to be loaded into memory at launch time and never unloaded. Using one or more nib files for each distinct tab makes it possible to load and dispose of the views in that tab on demand, potentially reducing your application's memory footprint.

How many nib files you use to specify the interface for a tab depends on the contents of the tab. The basic `UIViewController` class is designed to manage a single content view, which corresponds to a screen's worth of your application's custom content. (Thus, for simple interfaces with only one screen, you would need only one nib file.) For complex view controllers such as navigation controllers, you may need multiple nib files, each one representing the interface for a single distinct screen. At a minimum, every tab should have one nib file that you use to specify the root view for the tab.

To configure the root view for a tab, do the following:

1. Create a new nib file and configure it as described in "Using a View Controller as File's Owner of a Nib File" (page 61).
2. In the nib file containing your tab bar controller object, select the view controller for the appropriate tab. (You can also select the view controller by clicking the appropriate tab bar item once in the editor window.)
3. Open the Attributes inspector.
4. In the Nib Name field of the inspector, enter the name of the nib file (without any filename extension) you just created.
5. Save your nib file.

Configuring the Nib Name property of your view controller object in this manner is similar to creating your view controller programmatically using the `initWithNibName:bundle:` method. Whenever your application code requests its associated view, the view controller loads the specified nib file if it is not already in memory.

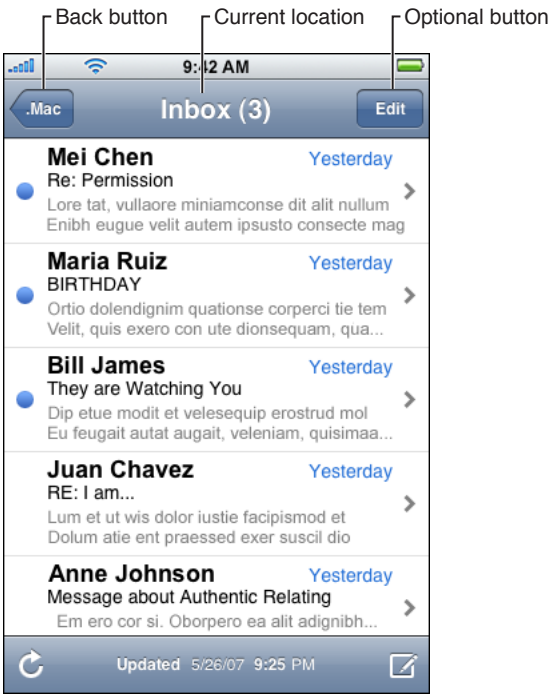
Specifying the nib file name also gives the view controller the option of releasing the contents of that nib file when they are not in use. It might do this in situations where the amount of free memory is running low. It can always reload the nib file again later if needed.

If you would rather create the views for the tab in the same nib file as your tab bar controller, simply select the tab in the tab bar controller editor window and drag the views into the provided space. The first view you add to the editor becomes the root view for your view hierarchy. If you intend to use multiple views, you should add a generic `UIView` object that can span the entire visible area and act as a parent for other views. Your remaining views would then be children of this root view.

Navigation Bars

Navigation bars are a visual element used in conjunction with a navigation controller object to navigate complex data hierarchies quickly and easily. The navigation bar displays the current location in the navigation hierarchy, provides an optional button for navigating back to the previous location, and provides an optional button for manipulating data at the current level. Figure 4-8 shows these components in the navigation bar used by the Mail application.

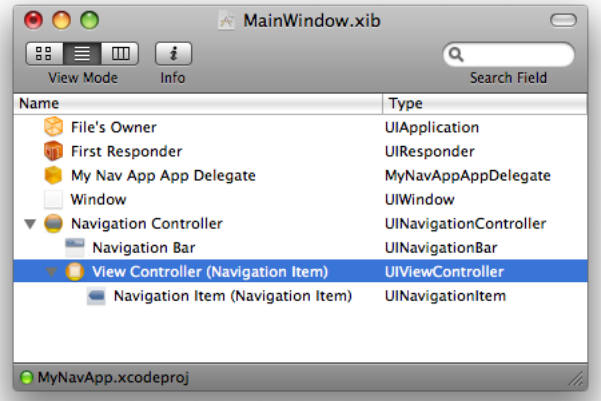
Figure 4-8 Components of a navigation bar interface



Although the navigation bar conveys contextual information to the user, the navigation controller provides the driving force behind displaying that interface and managing the view controllers that coordinate each screen's content view. Therefore, to configure a navigation interface in your nib file, you add a navigation controller object to the top level of your nib file.

When you add a navigation controller object to your Interface Builder document, you actually get several objects. In addition to the navigation controller, you get a navigation bar and a generic view controller. Interface Builder groups these objects together under the navigation controller, as shown in Figure 4-9.

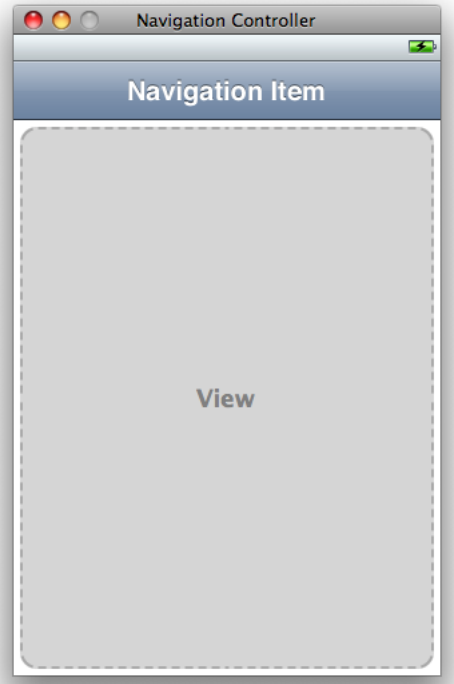
Figure 4-9 A navigation controller and its default set of objects



Important: A navigation controller object can have only two child objects: a navigation bar view and a view controller for managing its root view. Both of these objects must be present in order to initialize your navigation controller properly at runtime.

When you double-click a navigation controller object, Interface Builder displays the navigation controller editor window, which is shown in Figure 4-10. This editor window displays the child objects of the navigation controller object and is where you configure the root view of your navigation-based user interface.

Figure 4-10 Navigation controller editor window



The navigation controller editor displays a status bar (and other top bars and bottom bars) if the properties of that view controller indicate that one is present. These elements are not saved with your nib file and are provided to help you position your views and other content. You should configure the attributes of your navigation controller to match the presence of the status bar or other elements in your application window.

Although the navigation controller lets you edit the objects associated with the navigation-style interface, simply editing those objects does not make them appear in your application window. To display your navigation interface, you must programmatically retrieve the navigation controller's view and add it to your window as a subview. For example, if your application delegate has outlets referring to your navigation controller object and application window, its `applicationDidFinishLaunching:` method would need to look something like the following:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    [window addSubview:[myNavBarController view]];
}
```

If your navigation-style interface is used in conjunction with a tab bar interface, you would show the tab bar controller's view in your delegate's `applicationDidFinishLaunching:` method instead of your navigation interface. Whenever you combine navigation and tab bar interfaces, the tab bar interface must be used for the root level of your window. You can embed navigation controllers inside of a tab bar interface but you cannot do the reverse.

For information about configuring navigation bars and navigation controllers programmatically in your application, see "Navigation Controllers" in *View Controller Programming Guide for iOS*.

Configuring the Navigation Bar Items

Each navigation item in a navigation bar can be configured with a title, a prompt string (if any), and a string to use for the back button that returns the user to the screen containing that item. You can configure all of these properties using the Attributes inspector. In addition, double-clicking the title string of a navigation item in the controller's editor window lets you edit the title in place.

Configuring the Views for the Root Navigation Level

Each navigation level in a navigation interface has its own view controller and set of views for displaying the information at that level. The nib file containing the navigation controller object also contains the root view controller for hierarchy. There are two ways to configure the views associated with the root view controller:

- Use a separate nib file to specify the views. (Recommended)
- Embed the views in the same nib file that contains the root view controller.

Although you can embed the views for the root view controller in the same nib file as your navigation controller object, doing so requires those views to stay resident in memory when they are not in use. Storing the root views in a separate nib file makes it possible to load and dispose of those views on demand.

To create the view hierarchy for the root view controller using a separate nib file, do the following:

1. Create a new nib file and configure it as described in "Using a View Controller as File's Owner of a Nib File" (page 61).

   The class of the nib's File's Owner should be the same as your root view controller's class.

2. In the nib file containing your navigation controller, select the root view controller embedded inside the navigation controller object.
3. Open the Attributes inspector.
4. Set the class of the root view controller to your custom class.
5. In the Nib Name field of the inspector, enter the name of the nib file (without any filename extension) you just created.
6. Save your nib file.

Configuring the Nib Name property of your view controller object in this manner is similar to creating your view controller programmatically using the `initWithNibName:bundle:` method. Whenever your application code requests its associated view, the view controller loads the specified nib file if it is not already in memory. This gives the view controller the option of releasing the contents of the nib file when they are not in use and the amount of free memory is running low.

If you would rather create your the views for the root level of your navigation interface in the same nib file as your navigation controller, simply drag the desired views into the navigation controller editor window. The first view you add to the editor becomes the root view for your view hierarchy. If you intend to use multiple views, you should add a generic `UIView` object that can span the entire visible area and act as a parent for other views. Your remaining views would then be children of this root view.

Configuring the Views for Additional Navigation Levels

A navigation controller works by pushing new view controllers on to a stack as the user navigates further down the navigation hierarchy. Each new view controller manages the contents at the specified level of the hierarchy. Because the contents of the hierarchy are usually determined dynamically, you typically create the needed view controllers and push them on the navigation stack programmatically.

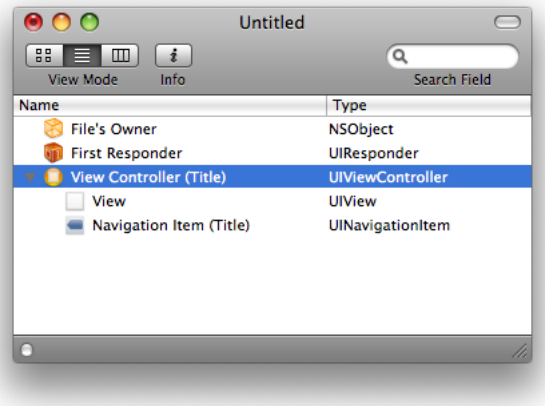
Each time you push a view controller, you must push the one that contains the desired interface for the target navigation level. If you use the same views in multiple levels of your navigation interface, you can create a new instance of one of your existing view controller classes to manage the content at the new navigation level.

Each new view controller you create must either load its views from a nib file or create them programmatically. Storing the views in a separate nib file makes it possible to load and dispose of those views on demand and is generally the recommended approach. For each distinct set of views you plan to use for your navigation interface, you must define a new subclass of `UIViewController` and a nib file to contain the views.

To create the nib file for one of your view controller subclasses, do the following:

1. Create a new nib file using the Cocoa Touch View template.
2. Add a view controller object to the top level of the nib file.
3. Set the class of the view controller to your custom class, as described in "Setting the Class of an Object" (page 139).
4. Drag a navigation item from the library and drop it onto the view controller object. (This embeds the navigation item in the view controller and causes it to be displayed in the view controller's editor window. It also sets the `navigationItem` outlet of the view controller to the item automatically.)
5. In the Interface Builder document window, drag the view that came with the template and drop it on the view controller to associate it with the controller. Your resulting nib file should look similar to the one shown in Figure 4-11.

Figure 4-11 Embedding the view in the view controller object



Making the view a child of the view controller automatically sets the `view` outlet of the view controller to the child view. If you do not want to embed the view in the view controller, you can also connect this outlet manually.

6. Double-click the view controller object to open its editor window.
7. Set the title of the navigation item to the default title for this navigation level.
8. Open the view and add any additional subviews to it to define your user interface.
9. Save the nib file.

To push a new view controller at runtime, create a new instance of your custom `UIViewController` subclass, initialize it with the nib file you created for it, and push it on the navigation controller stack. The new instance manages its views and content separate from any other instances you created previously. For information on how to push view controllers on the navigation stack, see "Navigation Controllers" in *View Controller Programming Guide for iOS*.

Mac OS X Interface Objects

This section describes the objects typically found in Cocoa and Carbon nib files. Objects in these nib files are intended for use in Mac OS X applications. For more information on the views and controls available for use in Mac OS X applications, see *Apple Human Interface Guidelines*.

Windows and Panels

Windows and panels provide the backdrop for drawing your application's content. Every application that has a graphical user interface has at least one window associated with it. Some applications may use multiple windows in different styles to present primary content (such as user data) or secondary content (such as inspectors or other utility windows).

In Cocoa applications, windows are based on the `NSWindow` class of the AppKit framework. In Carbon applications, views are based on the `WindowRef` data type.

Adding Windows to a Nib File

Windows and panels are top-level objects. In other words, you can add them only to the top level of your nib file. To add a window or panel to your nib file, do the following:

1. Open your nib file (or make it the active Interface Builder document).
2. Open the Library window and locate the desired window or panel object. (The library contains several different window and panel configurations. For tips on finding items in the Library window, see "Customizing the Library Window" (page 159).)
3. Drag the window or panel from the library and drop it either onto the desktop or onto your Interface Builder document window.

Dropping a window object onto the desktop adds that window to the top level of the active Interface Builder document window. The place where you drop the window becomes the window's initial position at load time. Once a window is associated with an Interface Builder document, dragging it around the desktop does not change its load-time position. Instead, dragging it around simply moves it out of the way and makes it easier to organize your workspace. To change the load-time position of the window from its initial setting, use the Size pane of the inspector window, as described in "Moving and Resizing Windows" (page 92).

Important: If you subsequently include an OpenGL view in one of your Cocoa windows, be sure to disable the window's "One Shot" option. The one-shot option deletes the window object when it is hidden or miniaturized to the dock. Unfortunately, destroying the window in this manner breaks the link between the OpenGL drawing context and the window, which means the view can no longer draw its contents. Disabling the one-shot option maintains the connection and ensures that the OpenGL view has a place to render its content.

When you double-click a window object, Interface Builder opens that window on the desktop (or makes it the active window if it is already open). The content area of the window represents the **design surface** and is where you build the appearance of your window.

Removing Windows from a Nib File

To remove a window from your nib file:

1. Select the window in your Interface Builder document window.
2. Press the Delete key (or choose Edit > Delete from the menu).

Removing a window removes the window, its contents, and any connections and bindings to other objects from the nib file.

Views

Views are the most commonly used objects in Interface Builder. The standard system views display data, respond to user input, and implement standard interface paradigms such as scrollable areas, controls, and many others. Applications can also create custom views and use them to display custom visual content and respond to user interactions.

In Cocoa applications, views are based on the `NSView` class of the AppKit framework. In Carbon applications, views are based on the `HIViewRef` data type.

Adding Views to a Nib File

You can add views to your nib file in the following ways:

- You can drag a view out of the library and drop it onto a window.
- In Cocoa nib files, you can drag the view to the top level of the nib file.

As you drag a view around your window, Interface Builder highlights different portions of the window to show you where the drop would occur. Upon dropping the view, the highlighted object becomes the parent of the dropped view. Interface Builder inserts the dropped view into the parent's view hierarchy and applies the parent's clipping rectangle. You can drag the view around the design surface to position it or change its parent view as needed.

You might place a view at the top level of your nib file in situations where you want to load only a view and not an entire window, such as when creating content for an accessory panel. When placing views at the top level of your document, be sure to assign them to an outlet of your File's Owner object so that you can dispose of them properly later. For more information about memory management and the objects in your nib file, see *Resource Programming Guide*.

In addition to adding standard system views to your window, you can also add views for which your Xcode project provides the view implementation. If you add such a view to your window, you must set the class of that view so that the proper object is created at load time. For more information about configuring custom views, see "Custom Views" (page 74).

Removing Views from a Nib File

To remove a view from a window or the top level of your nib file:

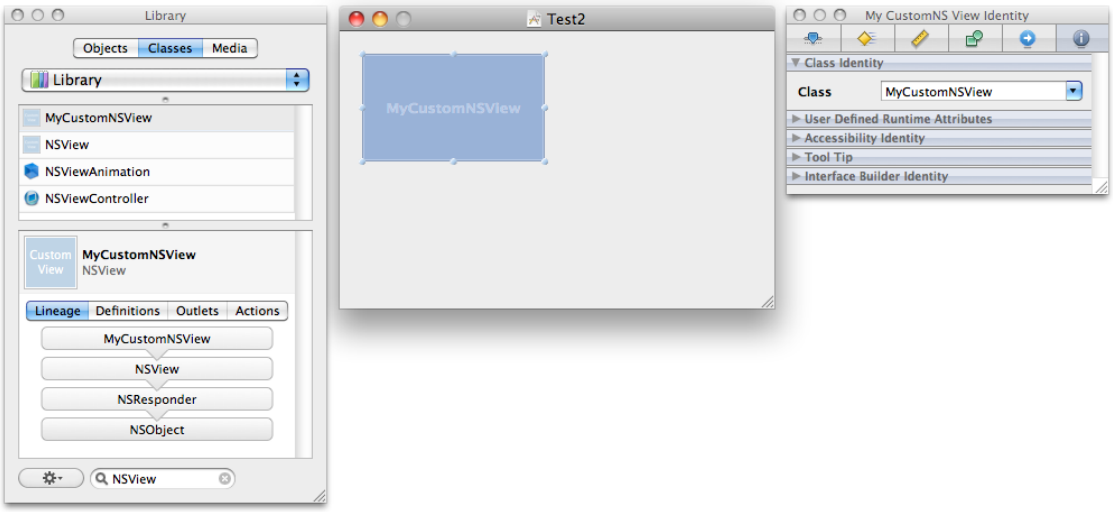
1. Select the view.
2. Press the Delete key (or choose Edit > Delete from the menu).

Removing a view removes the view, its child objects, and any connections and bindings to other objects from the nib file.

Custom Views

Although the standard system views and controls provide a variety of choices for building an interface, there are many times when you may want to provide customized behavior for your application. If your nib document is associated with a Cocoa project, Interface Builder is aware of any custom views declared in your source code. You can find your custom views listed in the Classes tab of the Library window. Dragging a custom view object to a window (or to the top level of your nib document) creates an unadorned rectangular area representing the space occupied by the object. Interface Builder sets the object's class name in the identity inspector, as illustrated in Figure 4-12. To learn about setting the class identity of a generic view object, see "Setting the Class of an Object" (page 139).

Figure 4-12 Custom NSView object



Because a custom view is one you are in the process of creating in your Xcode project, Interface Builder does not attempt to draw anything other than a generic box at design time. If you want to see the appearance of your custom view as it would appear at runtime, you must incorporate the finished view into an Interface Builder plug-in and load that plug-in. For more information about plug-ins, see "Using Plug-ins to Integrate New Objects into the Library" (page 167).

Controls and Cells

A control is just a type of view that handles a specific user interaction pattern and delivers messages to your application when that pattern occurs at runtime. When the user interacts with a control, the control object notifies your application in a way that is appropriate for the application type. Controls in a Cocoa application deliver a message to a target object. Controls in a Carbon application post an event to the application's designated event handler routine.

In Cocoa, the behavior of a control is provided by its cell object. Cells are lightweight objects that draw the control's contents and respond to user events within the control's frame. Prior to Interface Builder 3.0, controls and cells were treated as a single unit. In Interface Builder 3.0 and later, however, it is possible to manipulate cell objects independent of their owning control. You can select a cell and inspect its attributes (although in most cases, the attributes are shared with the control) and you can create connections to the outlets and actions of a cell independent of the the parent control object. The only time you might do this, however, is for the `NSMatrix` class, which can contain multiple cells. In most other cases, you should treat a control and its cell as a single entity.

You add controls to your window and remove them in the same manner as views. If you define custom cell classes, however, you can also substitute your custom cell for the standard cell of a control. To change the class of a cell, do the following:

1. Select the cell using the click-wait-click action on the control. (Alternatively, select the cell from the document window while it is in outline mode.)

2. Open the inspector window and choose the Identity pane.

3. In the Class field of the Class Identity section, enter the new class name for the cell. (If the class is defined in the associated Xcode project, Interface Builder automatically displays any outlets and actions defined by the custom cell class.)

When you save your Cocoa nib file, Interface Builder archives the custom cell information with the control. When the nib file is loaded at runtime, the nib loading code automatically creates your custom cell object instead of the standard one and assigns it to the control. This alleviates the need for you to set the cell object in your `awakeFromNib` method and also makes it easier to use different cell types for each control.

Custom Cells

Prior to Interface Builder 3.2, the library contained only specific `NSCell` subclasses such as `NSTextFieldCell`. This made it impossible to use a class that descended directly from `NSCell` in Interface Builder. The custom cell solves this problem.

A custom cell is analogous to a custom view. You can use a custom cell to represent any `NSCell` subclass.

Toolbars

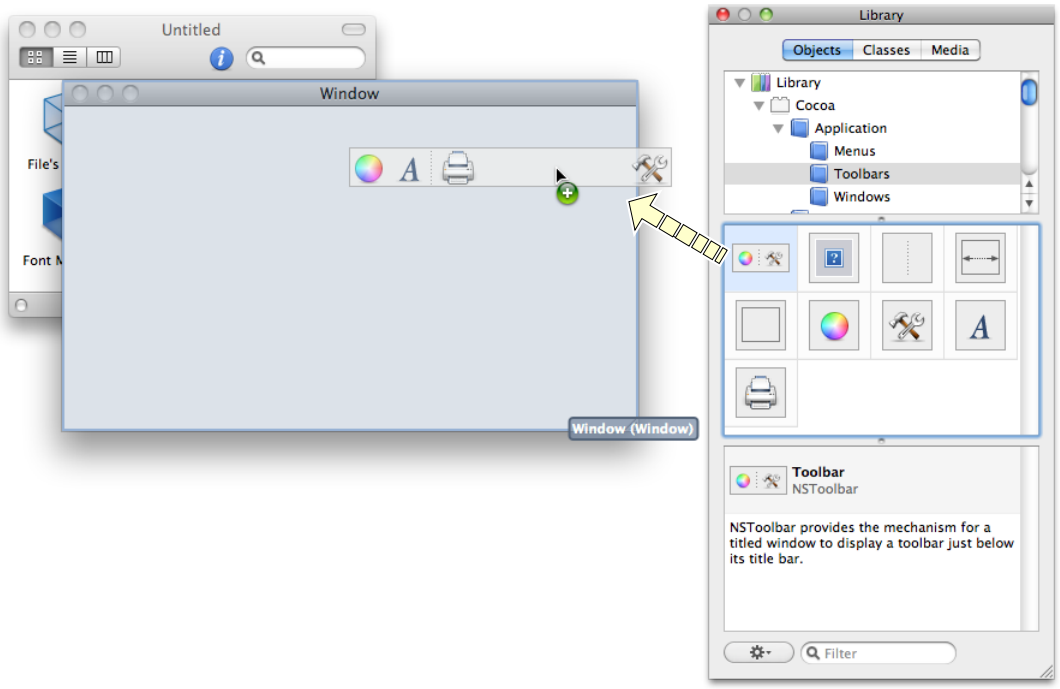
Toolbars provide the user with a convenient way to access frequently used commands without leaving the current window. A toolbar occupies the space between the title bar of a window and the window's content view. It spans the width of the window and displays one or more commands using a combination of icons and text. Toolbars are configurable by the user at runtime and developers use a very similar technique at design time to create the initial toolbar configuration in Interface Builder.

Note: Toolbars have always been available in Mac OS X but until version 3.0 were not configurable in Interface Builder. In Interface Builder 3.0 and later, you can add toolbars only to your Cocoa nib files and only if they are targeted at Mac OS X v10.5 and later.

Adding a Toolbar to a Window

To add a toolbar to a window, simply drag the toolbar item from the Library window and drop it on your window, as shown in Figure 4-13. The default toolbar comes configured with several standard items, including a color panel item, fonts panel item, print item, separator item, and fixed and variable spacer items.

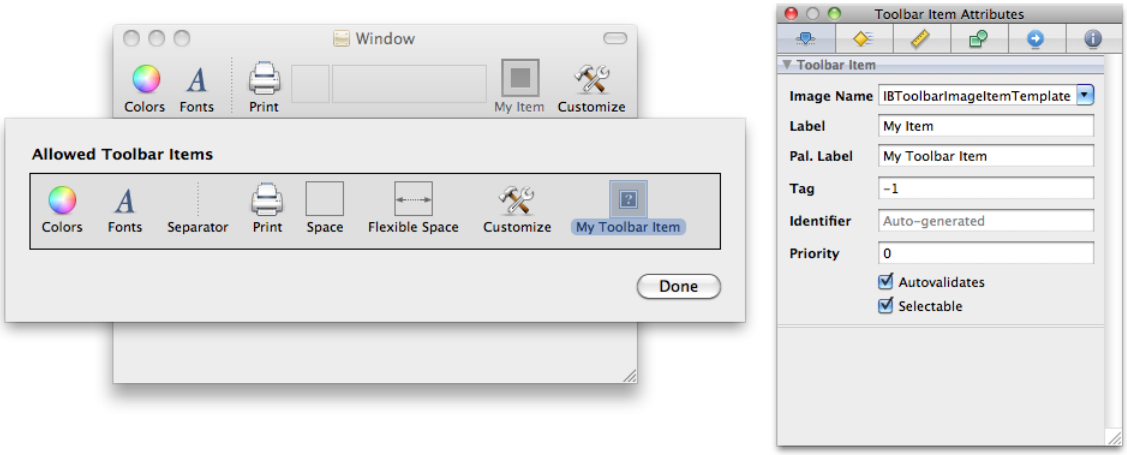
Figure 4-13 Adding a toolbar to a window



Customizing the Contents of a Toolbar

To customize a toolbar, double-click the toolbar in your window. Doing so opens up the toolbar customization sheet, which you can use to add, remove, and rearrange items in the toolbar (Figure 4-14). This sheet is similar to the customization panel the user sees when configuring a toolbar at runtime and works in much the same way.

Figure 4-14 Customizing a toolbar



To add a new custom item to the toolbar, do the following:

1. Drag an Image Toolbar Item from the Library window and drop it onto the customization sheet.
2. Select the Image Toolbar Item and open the inspector window.
3. In the Attributes pane of the inspector window, set the following attributes:
    - The name of the image to use for the item (Image Name attribute)
    - The item text as it appears in the toolbar (Label attribute)
    - The name of the item as it appears in the user customization sheet (Pal. Label attribute)
4. Position the item in the customization sheet by dragging it to the desired location.
5. If you want the item to be part of the default toolbar, drag it to the toolbar.
6. Click Done to close the customization sheet.

Note: You must add toolbar items to the customization sheet before you can add them to the toolbar itself.

The image name you specify should correspond to an image resource from your associated Xcode project. When specifying the image name in the inspector window, you do not need to include the filename extension. Your images should be sized appropriately for the size of the toolbar you plan to use. Regular size toolbars use images that are 32 by 32 pixels. Small toolbars use images that are 24 by 24 pixels. If an image file contains multiple image representations, the Image Toolbar item chooses the one that is closest to the toolbar size and then scales it as needed.

Removing Items from a Toolbar

To remove an item from the toolbar, drag it off of the toolbar or the customization sheet. Dragging an item off of the toolbar removes it from the toolbar but not from the customization sheet. Dragging an item off of the customization sheet removes it from both places.

Responding to Toolbar Events

To respond to user clicks in toolbar items, you must configure each item to send its action message to an appropriate target object. You configure the toolbar actions using the connections panel. Control-clicking a toolbar item opens the panel and lists the actions available for the toolbar. Most items have just one action that you use to connect the toolbar to a target object. In addition to configuring the item's action, you can also set its enabled state using Cocoa bindings. For more information about creating connections, see "Connections and Bindings" (page 119).

Removing a Toolbar from Your Window

To remove a toolbar from your window:

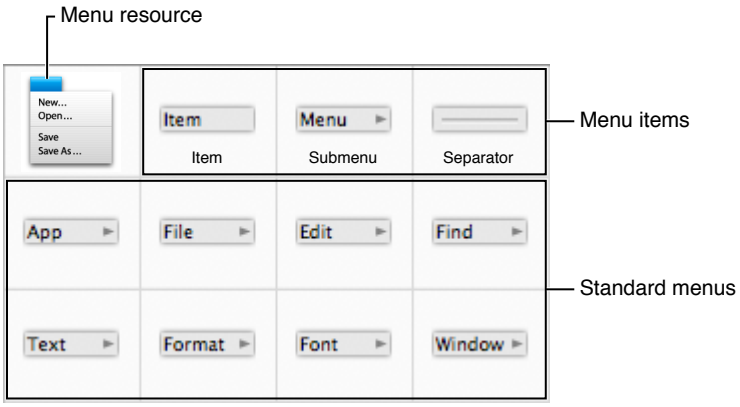
1. Select the toolbar. (The customization sheet must be closed.)
2. Press the Delete key (or choose Edit > Delete from the menu).

Menus and Menu Items

Menus are an important part of Mac OS X applications. Every application displays a menu bar with commands for manipulating the application and its data. Menus can also appear in more localized parts of your application. For example, applications often use context-sensitive menus to store commands relevant to the object underneath the cursor. All of these types of menus are represented in Interface Builder using menu resources.

A menu resource is a special type of object that holds a collection of menu items. Menu items represent the actual commands included in the menu. You can also insert submenu items in situations where you want to create a hierarchical organization for your menu items. Figure 4-15 shows the menu objects that appear in the Library window. In addition to the basic menu resource and menu item objects, the library contains preconfigured menu resources representing standard system menus, such as the Font and Window menus.

Figure 4-15 Menu objects in the library



The following sections describe the standard ways to create and organize menu bar and menu resources. For information on how to tie those menu items to specific parts of your application code, see "Connecting Menu Items to Your Code" (page 127).

Creating and Modifying Menu Bars

A menu bar resource is a special type of resource used to store the application menus. Because an application has only one menu bar, you do not create menu bar resources by dragging them out of the Library window. When you create a new Xcode project, the application's main nib file already comes with a menu bar resource that you can modify.

If you create your project from scratch or somehow delete the main nib file from your Xcode project, you can create a new nib file with a menu bar resource. Interface Builder provides template nib files for Carbon and Cocoa applications that include a preconfigured menu bar resource. When you create your nib file, select either the Application or Main Menu template from the template picker.

Once you have a menu bar resource, you can add and remove menus and menu items accordingly. To add, remove, and manage menus in a menu bar, you do the following:

- To add a new custom menu to the menu bar, drag a Submenu Menu Item from the library and drop it on the menu bar. (You must use the Submenu item and not the Menu resource when adding menus to a menu bar.)

- To add a standard Mac OS X menu (File, Edit, Window, and so on) to the menu bar, drag the appropriate menu object from the library to the menu bar.
- To select a menu, click the menu title, wait, and then click the title again so that the menu is selected but its underlying menu is closed.
- To remove an existing menu, select the menu and choose Edit > Delete (or simply press the Delete key). (The menu must be closed before you can delete it.)
- To change the name of a menu, double-click its title to edit the name in place, or select the menu and change its title in the inspector window.

When adding new menus to a menu bar, if an existing menu is already open, hold the mouse over the menu bar for a few seconds until the current menu closes. At that point, you can drop the new menu on the menu bar. For information about modifying the individual items in a menu, see "Creating and Modifying Menus" (page 80)

Creating Custom Menu Resources

Menu resource objects live at the top level of your Interface Builder document and are not the same as a menu bar resource. You cannot use a menu resource to specify menus in the menu bar. Instead, you use menu resources for menus in places other than the menu bar. For example, if a control has a context-sensitive menu, you would use a menu resource to provide the contents of that menu.

To add a menu resource to your nib file, simply drag it from the library and drop it on your Interface Builder document window. Double-clicking the menu resource opens up an editor window containing some default menu items. You can change the names of these items, add new items, or delete the existing items to form your menu. For more information about building the contents of your menus, see "Creating and Modifying Menus" (page 80).

Creating and Modifying Menus

Menu bar and menu resources organize the menu items associated with your application. Menu bar resources are created with your nib file but you can add new menu resources to your nib file to store context-sensitive menus. Once you have a menu resource in your nib file, you can move or remove existing items and add new submenus and menu items from the library.

To add, remove, and rearrange menu items in a menu, do the following:

- To add an item to a menu, drag the item from the library and drop it on the menu. If the target menu is closed, hold the item over the menu title (if in the menu bar) or menu resource until the menu opens automatically. As you drag the item, Interface Builder shows you the target location for the item in the menu.
- To select a menu item, click the menu item title. If the item represents a submenu, you must click the item again to close the submenu and select the item.
- To remove a menu item, select it and choose Edit > Delete (or simply press the Delete key).
- To rearrange items in a menu, drag them to the desired location in the menu.
- To embed an existing menu item in a new submenu, select the item (or items) and choose Layout > Embed Objects In > Submenu.
- To change the title of an item, double-click its title string to edit the title in place or select the item and change its title in the inspector window.

- To edit the item's attributes, select the item and make the changes in the inspector window.

Tip: To quickly add several new items to a menu, add the first item by dragging it from the Library window, select that item, and choose Edit > Copy. You can then paste multiple copies of the item in quick succession below the selected item.

Collection Views

In Mac OS X v10.5 and later, you use collection view objects to manage a grid of `NSView` objects in a Cocoa application. You can use a collection view in situations where using `NSCell` objects in a matrix would be insufficient. Because they are full-fledged views (and not cells), the views in a collection view can handle events, perform mouse tracking, and do anything else a view could normally do without having to write a lot of custom code.

When you drag a collection view to one of your windows, Interface Builder creates two additional objects at the top level of your nib file:

- A generic `NSView` object, which acts as a template view for the items in the collection.
- An `NSCollectionViewItem` object, which is a controller that manages the relationship between a model object and a single view in the collection.

These two objects serve as the prototype for each item in the collection view. The collection view uses the prototype view as a template for creating each new view in the collection. Each time it creates a new view at runtime, it also creates the corresponding `NSCollectionViewItem` object to manage that view.

Configuring a collection view involves configuring the template view and setting up bindings between it and the corresponding controller object. The contents of your view can be anything you want. You could even use a tab-less tab view to create different interfaces for different views. The controls in your view are then bound to the `NSCollectionViewItem` object through its `representedObject` binding.

To provide the actual data behind the `representedObject` binding, you must set the content of the collection view object itself (not the `NSCollectionViewItem` controller). You can do so directly using the `setContent:` method of the `NSCollectionView` class, or you can set the content using Cocoa bindings. The content attribute of `NSCollectionView` stores an array of data objects, each of which contains the custom data to be displayed by one view in the collection. As it populates the grid of views, the collection view automatically associates the items in this array with the corresponding `NSCollectionViewItem` objects used to manage the grid views. Each collection view item then exposes its particular piece of data through its `representedObject` attribute. If the custom data objects you set as the content for your collection view are KVC and KVO compliant, you can bind to their exposed attributes. For example, if your custom data object exposed a `name` attribute, you could bind a text field in your template view to the collection item controller using the key path "`representedObject.name`".

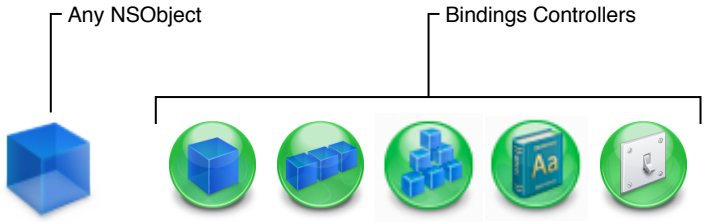
Controller Objects

In addition to visual objects, Cocoa nib files can include any type of custom object (including non-visual objects) needed by an application. The ability to include any instance of `NSObject` is typically used as a way to facilitate the Model-View-Controller design pattern used by Cocoa applications. The non-visual objects in a nib file act as controllers for the visual objects.

Including controller objects directly inside a nib file has many advantages over creating them separately. Cocoa nib files are typically managed by a single object (known as the File's Owner) that exists outside of the nib file itself. However, it is often convenient to use additional controller objects to manage portions of the nib file contents. In particular, Cocoa bindings make extensive use of custom controller objects to manage the interactions between the views and the application's underlying data model.

Figure 4-16 shows the types of custom objects you can add to a Cocoa nib file. The generic `NSObject` type lets you create an instance of any object that descends from the `NSObject` class. In addition to this type, Interface Builder provides specialized objects representing specific `NSController` subclasses for managing Cocoa bindings. The Core Data objects provide a way to incorporate managed object contexts and interfaces for your Core Data entities into your nib files.

Figure 4-16 Controllers and custom objects



To add a generic `NSObject` instance to your nib file, simply drag the Object item from the library and drop it on your Interface Builder document window. (Custom objects cannot be embedded inside other objects in your nib file; they must reside at the top level of the Interface Builder document window.) After adding a generic object to your nib file, you should specify the intended class of that object right away. Doing so lets you connect any outlets or actions of that object to other objects in your nib file. For information on how to set the class of an object, see "Setting the Class of an Object" (page 139).

Like generic objects, you must drag controller objects to the top level of your Interface Builder document window. Unlike generic objects, you should not change the class of a bindings controller object. For information on how to configure the bindings controllers, see "Using Cocoa Controller Objects" (page 134).

View Controllers

Applications with only one window often change their content by changing the content views displayed in that window. The **content view** is the portion of a window that you use to display your application's custom content. The content view is not necessarily a single view but may in fact comprise several views embedded inside a parent (or root) view. Management of the content view is the responsibility of a view controller object, which coordinates the movement of that content on and off the screen. Because it is a controller, you can also add custom logic to your view controller classes to manage interactions between the content view and your application's data model.

Interface Builder provides a generic view controller object that you can use for managing individual views. The most common way to use a view controller is to instantiate it inside your nib file, but you can also use it as the File's Owner of the nib file.

Adding a View Controller to Your Nib File

To add a view controller to the top level of your nib file, drag it from the objects tab of the library into the document window. The view controller is configured a bit differently from many other objects—it has a nib name property that identifies a second nib to load at runtime.

Using a View Controller as File's Owner of a Nib File

To use a view controller as the File's Owner of your nib file, do the following:

1. Create a new nib file using the Cocoa View template and configure it as follows:
    a. Set the class of the File's Owner placeholder object to the class of your custom view controller.
    b. Connect the view outlet of the File's Owner to the view provided by the template.
2. Open the view and add any additional subviews to it to define your user interface.
3. Connect any additional outlets and actions.
4. Save the new nib file and add it to your Xcode project.

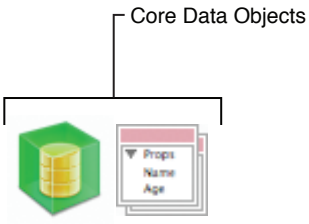
Nib files configured in this way are associated with the corresponding view controller class at runtime. When you create a new instance of your view controller, you pass the name of your nib file to the `initWithNibName:bundle:` method of the `NSViewController` class. The first time your application asks the view controller for its view, the view controller loads the associated nib file and returns the view attached to its `view` outlet. The view controller continues to manage the contents of the nib file internally, purging the views as needed during low-memory conditions and reloading them later as needed.

Nib files configured in this manner are often used to implement tab bar and navigation-style interfaces. The content view for each distinct screen is stored in its own nib file and associated with a view controller class. The navigation and tab bar controllers then manage the loading and unloading of each view controller and its associated nib file.

Core Data Objects

Interface Builder works with Xcode to support the creation of user interfaces based on Core Data entities. After creating your data model in Xcode, all you have to do to create an interface for that entity's data is drag a Core Data Entity object into one of your windows. Interface Builder steps you through the creation of your interface using a series of design panels, which let you pick what you want to display and show you the appearance of the resulting user interface. Upon completion, Interface Builder adds the appropriate views to your nib file and connects them to the corresponding entities in your data model. If you want to manage the data in your Core Data database yourself, you can also add managed object contexts to the top level of your nib file. Figure 4-17 shows the managed object context and entity items as they appear in the library.

Figure 4-17 Core Data objects



For more information about creating user interfaces for Core Data applications, see *Xcode Tools for Core Data*. For more information about creating Core Data programs, see *Core Data Programming Guide*.

Formatter Objects

Cocoa applications can use formatter objects to automatically format data displayed in text-based controls. Cocoa provides built-in formatters for number and date values. The formatters themselves use a set of predetermined rules to format the values they receive into something more understandable for the user. For example, you can use a number formatter to display a number as a currency value, with the appropriate separator characters and monetary indicator included in the resulting string.

For information about how to apply formatters to your Cocoa objects, see "Applying Formatters" (page 109).

Placeholder Objects

In Cocoa and Cocoa Touch nib files, a **placeholder object** is a placeholder for an object that is used in a nib file but not stored in it. (Placeholder objects are not available in Carbon nib files.) It is important to remember that a nib file is simply a set of objects stored in an archival format. When you load a nib file, the nib-loading code unarchives the objects, effectively reconstituting them inside your application. Without placeholder objects, however, those objects would be isolated from the rest of your application code. The presence of placeholder objects inside a nib file provides a bridge between the code you define in your application and the objects in your nib file.

The following sections describe the standard placeholder objects that are provided automatically by Interface Builder where appropriate. For information about how to create custom placeholder objects in iOS nib files, see "Custom Placeholder Objects" (page 86). For additional information about how placeholder objects are replaced by actual objects at load time, see *Resource Programming Guide*.

File's Owner

File's Owner is the most commonly used placeholder object in nib files and is supported by both Cocoa and Cocoa Touch nib files. In essence, the File's Owner placeholder is the main bridge between your application and the contents of a nib file. In a nib file, the File's Owner placeholder is a placeholder for an object that you plan to specify when you load the nib file. The object does not exist in the nib file itself and is not created when the nib file is loaded.

When you load your nib file into memory, the nib-loading methods expect you to pass along an object that you want to designate as the nib-file owner. The class of the object you specify must match the class of the File's Owner placeholder in the nib file. As the nib file is loaded into memory, the nib-loading code substitutes the object you provide for any references to the File's Owner placeholder in the nib file. This substitution results in your object's outlets and actions being automatically connected to the objects inside the nib file.

You can designate any of your application objects as the File's Owner of a nib file. Typically, the File's owner object is a controller object that manages the interactions with the views and other controller objects inside the nib file. Table 4-1 lists some of the standard classes that are commonly used to represent File's Owner in applications.

Table 4-1 Typical classes used for File's Owner

Class	Description
`NSDocument`	Cocoa document-based applications store the document window and other required interface objects in a nib file. The File's Owner of this nib file is traditionally the document object itself (a subclass of `NSDocument`). When the user requests a new document, Cocoa creates the `NSDocument` object and uses its associated `NSWindowController` object to load and manage the associated nib file contents. Each document in the application receives its own custom copy of the nib file objects so as to avoid unwanted interactions between documents.
`NSWindowController`	In Cocoa applications, it is common to use a subclass of `NSWindowController` to manage custom alert panels, modeless panels, and other windows. Window controllers provide a great deal of automatic management for nib files and are especially useful when your nib file contains only one window and perhaps some supporting objects or controls.
`NSViewController`	In Cocoa applications, a view controller manages custom accessory views and other view-based content. For example, printing accessory panels rely on the use of a view controller to manage the printing behavior.
`UIViewController`	In iOS applications, separate nib files are often used to manage the content view for each distinct screen's worth of content. The manager for this content view is a custom `UIViewController` object, which also provides automatic nib-loading and purging support.
Any custom `NSObject` subclass	If you want to manage a nib file manually, you can use practically any object you like. You might use a custom subclass in situations where you want more control over the management of the nib-file objects. It is up to you to define the relationships between this object and the objects in your nib file.

To configure the File's Owner placeholder, select it in the Interface Builder document window and open the Identity inspector. In the Class field of the Class Identity section, set the value to the corresponding class in your application. Once the class is set, Control-clicking the File's Owner object displays the outlets and actions defined by that class. You can use these outlets and actions to connect other objects to File's Owner. You can use File's Owner as a target for your bindings. For information about configuring and connecting to File's Owner, see "Connections and Bindings" (page 119).

First Responder

Cocoa and Cocoa Touch applications use the First Responder placeholder object as a placeholder for the first object in the responder chain. The First Responder typically corresponds to the currently selected object or the object with the current focus in the frontmost window. You use the First Responder as a target for any messages that operate on the current selection or need to be handled by the frontmost window or document. For example, if you wanted a menu command to be handled by the frontmost window, you would dispatch that command to the First Responder object.

The First Responder is not required to respond to a given message. If the First Responder does not respond to a given message, Cocoa passes that message to other objects in the responder chain until one does respond. If no object responds, the message is ignored. It is the responsibility of the objects in your responder chain to implement action methods for messages you want to handle.

The First Responder placeholder initially displays all of the actions that are either supported natively by Cocoa or defined in your Xcode source files. If you want to add action methods to the First Responder that are not present in either of these locations, you can do so using the Identity inspector. The First Responder Actions section of the Identity inspector includes a plus (+) button for adding new actions to the available list. For information on how to make connections to the First Responder placeholder, see "Establishing Connections to the First Responder" (page 126).

For more information about event handling and the responder chain in Mac OS X applications, see *Cocoa Event-Handling Guide*. For information about the role of the responder chain in iOS applications, see *iOS Application Programming Guide*.

Application

In Cocoa nib files, the Application placeholder object gives you a way to connect the outlets of your application's shared `NSApplication` object to custom objects in your nib file. The default application object has outlets for its delegate object and, in Cocoa applications, the application menu bar. If you define a custom subclass of `NSApplication`, you can connect to any additional outlets and actions defined in your subclass.

At load time, the nib-loading code automatically replaces the application placeholder object with the shared application object from your application. You do not need to specify this object explicitly when loading your nib files.

Custom Placeholder Objects

The objects in an iOS application's user interface tend to be much more intertwined than they are on other platforms. Many iOS application interfaces are organized around view controller objects, which manage a hierarchy of views. These view controller objects may then have to interact with several other controllers and views that are already present in the running application. To accommodate the presence of these extra controllers, Interface Builder supports the specification of custom placeholder objects for iOS applications.

You configure custom placeholder objects in the same way that you configure a generic object. You add the placeholder object to the top-level of your Interface Builder document, set its class, assign it a name, connect its outlets and actions, and save it with the rest of your document to a nib file. At load time, however, the nib-loading code does not instantiate a new instance of your placeholder object. Instead, your call to load

the nib must be accompanied by a pointer to the real object that represents the placeholder. The nib-loading code then swaps in the real object for the placeholder and proceeds to reconnect the outlets and actions of that object. You can include any number of custom placeholder objects in your nib file.

The key to replacing a placeholder object at load time is in naming the object. The nib-loading code requires you to specify both the replacement object and the name of that object in a dictionary. You specify the name of the object in the Name field of the Identity inspector. Placeholder object names in your nib file must be unique in order to ensure the correct objects are connected to the right placeholder. The rest of the placeholder object configuration is the same as it is for other objects. You set the class as described in "Setting the Class of an Object" (page 139) and configure the actions and outlets as usual.

For information about how to replace placeholder objects with real objects at load time, see *Resource Programming Guide*.

Finding Objects in a Nib File

Interface Builder provides several tools to make it easier to find objects in the Interface Builder document window or on the associated design surfaces. These tools are convenient in situations where your nib file has many objects or those objects are deeply nested inside a view hierarchy.

- To find objects in the document window:
  - Type the name of the object in the search field.
  - Select the object on the design surface and choose Tools > Reveal in Document Window.
- To find objects on the design surface, select the object in the document window and choose Tools > Reveal in Workspace.

Tips for Organizing Your Nib Objects

The following sections provide some tips for organizing your user interface into nib files.

Factor Your Interfaces Appropriately

Deciding what objects go into a nib file is an important consideration. Because nib files are resource files, it is generally better for performance if a nib file contains only the immediately needed objects. If your nib file contains several unrelated windows and menus, you pay an upfront memory penalty when loading that nib file. Any objects in a nib file that are not put to immediate use remain in memory and increase your application's footprint. In addition, if an object is paged to disk before it is even used, you pay a penalty for having to load it from disk twice.

Cocoa nib files are typically organized around a single window or menu resource. Cocoa Touch nib files are typically organized around a single window or content view resource. All other objects in the nib file are those that are needed to support the given window, menu, or view resource. For example, the nib file might contain any additional controller objects needed to manage the resource.

If you already have nib files with many unrelated resources, you can use the built-in refactoring tools to break your nib file up into smaller, independent nib files. For information about using the refactoring tools, see "Refactoring Your Nib Files" (page 53).

Use the Embed Commands

Interface Builder makes it easy to embed existing views in your interface inside other views quickly. Rather than dragging a new container view into your window and copying your objects into it, you can use the Layout > Embed Objects In menu instead to create the desired container relationship. This option is the most effective way to create embedded relationships for most container views.

Interface Layout

Interface Builder helps you experiment with different user interface designs by providing you with tools that make it easy to add and change the layout of the windows and views that comprise your interface. You make most of your layout changes directly using the mouse. You start by dragging objects out of the library and into your Interface Builder document. You then use the mouse to position those objects, resize them, and edit their attributes either inline or from the inspector window.

In situations where the mouse does not offer the level of control you need, Interface Builder provides other tools to give you precise control over the size and position of individual objects or groups of objects. The following sections describe the layout tools available for you to use along with guidelines on how to use those tools effectively.

Layout Tools

Table 5-1 lists the layout tools that are available for you to use in Interface Builder.

Table 5-1 Tools to help with laying out user interfaces

Tool	Description
The mouse	The mouse is the tool you use to make coarse adjustments in the position or size of an item. As you move the mouse, the cursor image changes to reflect the action that would occur if you clicked at the current point. For example, a hand cursor indicates that you can drag an item.
Size inspector	The size inspector contains field for fine tuning the size and position of items. It also contains tools for aligning and positioning multiple objects relative to one another.
Guides	Guides provide alignment information for items on the design surface. You can use the automatic layout guides, custom fixed guides, and instantaneous guides. For information about using guides, see "About Alignment Guides" (page 96).
Outline view mode	In outline view mode, an Interface Builder document displays the hierarchy of the objects in that document. While in this mode, you can rearrange view hierarchies by dragging and dropping objects.

Moving and Sizing Objects

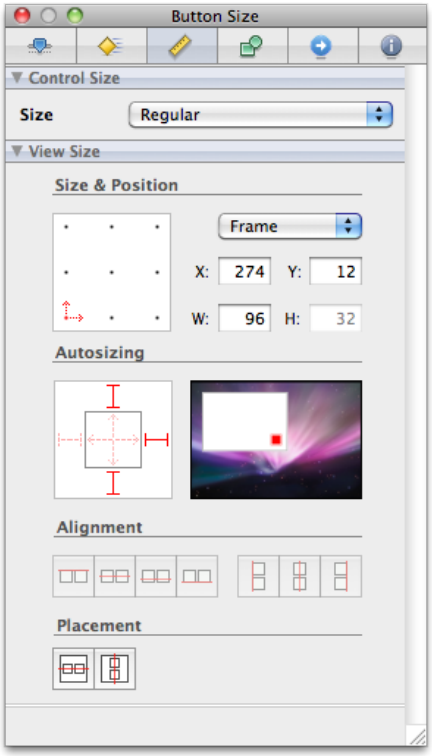
Interface Builder provides several different techniques for moving and resizing windows and views. These techniques are discussed in the sections that follow.

Moving Views

There are several different ways to change the position of a view relative to its parent view:

- If the view is unselected, click and drag it to its new location.
- If the view is selected, move the cursor over the view until the open hand cursor appears. With that cursor visible, click and drag the view to reposition it.
- If the view is selected, change the values in the X and Y fields of the inspector window (Figure 5-1).
- If the view is selected, use the arrow keys to nudge it one pixel in any direction.

Figure 5-1 The Size pane of the inspector



Moving and resizing operations do not automatically change a view's relationship to its parent view. If you click and drag a view that is embedded in another view, Interface Builder moves the dragged view within its parent view's frame and clips the dragged view appropriately. For information on how to disassociate a view from its parent so that you can drag it elsewhere in a window, see "Changing a View's Parent to Another View" (page 102).

When moving views using the inspector, you can use the grid control (in the Size & Position section) to position the view relative to different points in its frame. For example, instead of specifying the position of the view's lower-left corner, you could specify the position of its center. To do that, you would click the middle point in the grid control to establish that the X and Y values should reflect the center point of the view's frame. Changes to those values would then change the position of the view so that it's center point matched the new values.

To nudge an object by more than one pixel at a time, you can either hold down the arrow key or use it in combination with the Shift key. Holding down an arrow key nudges the selected view repeatedly at the same speed as the current key repeat rate. Holding down the Shift key increases the nudge increment from one pixel to five pixels.

For information on moving windows, see "Moving and Resizing Windows" (page 92).

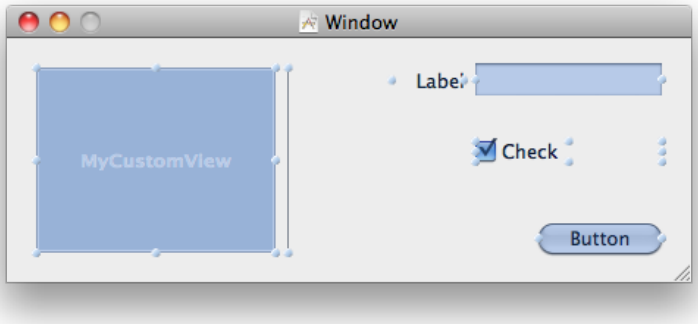
Resizing Views

After selecting a view, there are two ways to resize it:

- Drag one of its selection handles.
- Change the width and height values in the size inspector.

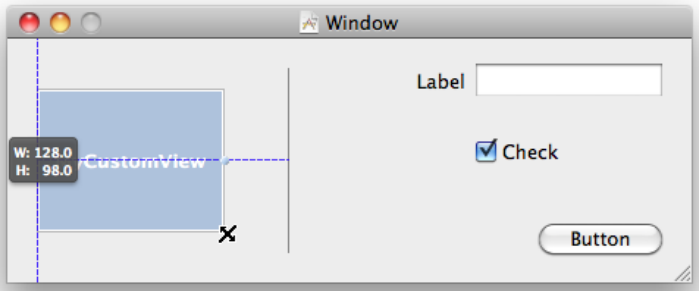
Figure 5-2 shows the selection handles created for several different types of views in Mac OS X. You can use a view's selection handles to resize that view in any of the available directions. Moving the mouse over a selection handle changes the cursor to indicate that a click would initiate a resize operation.

Figure 5-2 Examples of selection handles



During a resize operation, the current width and height of the view (in points) are displayed. If you press the Shift key while diagonally resizing a view with the mouse, Interface Builder maintains the initial aspect ratio of the view. Figure 5-3 illustrates both of these features.

Figure 5-3 A resize operation



If a control is too small—for example, if the control's title is too wide to be displayed—Interface Builder draws a plus icon over the control. Clicking the plus icon sizes the control to fit. If you want to disable this behavior, uncheck the Show Clipping Indicators menu item in the Layout menu.

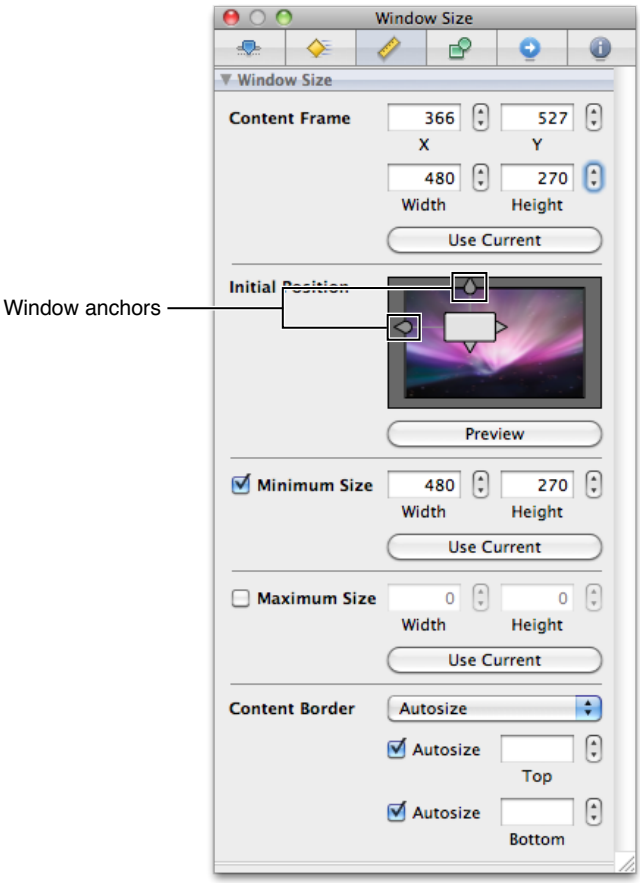
Moving and Resizing Windows

When a window is loaded from a nib file, the nib-loading code uses the window's frame rectangle to position it on the user's screen. Because the screen you use at design time may be a different size than the user's screen, Interface Builder disassociates the load-time position of the window from its position on your screen while you work on it.

When you drag a window from the library to the desktop, Interface Builder adds the window to the active nib file and uses the drop location as the initial load-time position of the window. Once it is in your nib file, however, dragging the window around your screen does not change its load-time position. The only way to change the window's load-time position is using the window size inspector.

Figure 5-4 shows the size inspector for a window in a Cocoa application. To change the window's load-time position, change the X and Y values in the Content Frame section of this inspector. (You can also click the Use Current button to use the window's current position on your screen as the new load-time position.) To compensate for variances in the user's screen size, you can use the provided window anchors to assign a fixed distance to the space between the window and the corresponding edge of the screen. Because a window's origin is in the lower-left corner of its frame, specifying the window position based solely on the window origin would result in the window being positioned differently on different size screens. Window anchors ensure the window appears at the same relative location regardless of screen size.

Figure 5-4 Size inspector for windows



To change the size of a window, you can do any of the following:

- Click and drag the resize handle in the lower-right corner of the window.
- Change the width and height values in the Content Frame section of the size inspector.

Pressing the Shift key while dragging a window's resize handle constrains the resizing operation to the current dragging direction. A horizontal drag keeps the window's height fixed but changes the width, while a vertical drag resizes only the window height. A diagonal drag resizes the window proportionally in both the horizontal and vertical directions.

Note: When resizing a window using its resize handle, Interface Builder does not change the size or position of the window's contained views by default. You can toggle this behavior in situations where you want any contained views to scale up or down based on the new window size. For more information, see "Design-Time Resizing Modes for Windows" (page 95).

In addition to setting the window's size and initial position, you can use this inspector to configure both a minimum and maximum size for the window. Setting these values is recommended, especially if making your window too small or too large might cause controls to be obscured or the layout of your window's contents to change drastically.

You can also use this inspector to modify the window's content border. If you choose autosize, Interface Builder uses a standard content border. In some cases, you can customize this border by entering the desired inset value in the bottom or top fields. This section maps directly to two methods in the `NSWindow` class:

```
setAutorecalculatesContentBorderThickness:forEdge:
setContentBorderThickness:forEdge:
```

The `NSWindow` API supports modifying the content border thickness for all four window edges, but the current implementation only supports setting the bottom, and in some cases the top. Because of this, Interface Builder only shows inspector elements to configure the top and bottom inset values.

Changing the Size of a Control

Mac OS X supports different sizes for controls to accommodate different size interfaces. The size of a control determines both the size of the control itself and how much space must be left between it and other controls. For many controls, control size also imposes a fixed height or width. For example, buttons typically have a variable width but a fixed height that is determined by the control size.

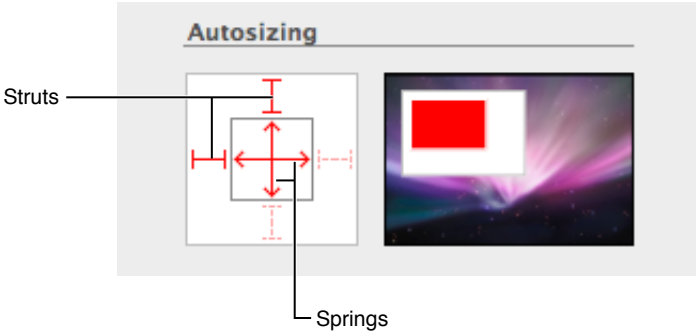
To set the size of a control, use the size inspector (Figure 5-4 (page 93)). In the Control Size section, select the desired size from the pop-up menu. All Mac OS X controls support the regular size but not all controls support other sizes. Because the controls in a single window should all be the same size, you should check to see whether the correct size is available for all controls you plan to use before starting your design.

Setting a View's Autosizing Behavior

When the user' resizes a window, the system automatically resizes the views inside that window according to their autosizing rules. (Autosizing also occurs for views, such as split views, that themselves can change dynamically in size.) All iOS and Mac OS X applications support autosizing rules so as to avoid the need for you to write code to resize those views manually. The **autosizing rules** determine whether a view expands or contracts to fill the new space and whether there is a fixed or variable amount of space between its frame and the frame of its parent view.

In Interface Builder, you configure the default autosizing rules for your views using the size inspector, as shown in Figure 5-5. The springs and struts in the autosizing control define the selected view's relationship to its parent frame. A **spring** causes the view to resize itself proportionally based on the width or height of its superview. A **strut** causes the view to maintain a fixed distance between itself and its superview along the given edge. As you change the springs and struts in the inspector, the animation to the right of the control provides a visual depiction of the resulting autosizing behavior of the view.

Figure 5-5 Configuring the autosizing rules of a view



In addition to configuring your view's autosizing behavior in Interface Builder, you can also configure it programmatically. For UIView objects, you do so using the `autoresizingMask` property of the view. For `NSView` objects, you use the `setAutoresizingMask:` method of the view. In Carbon, you use the `HIViewSetLayoutInfo` function.

Important: In a Cocoa nib file, if you do not set any springs or struts for your view in Interface Builder but then do use the `setAutoresizingMask:` method to add autosizing behavior at runtime, your view may still not exhibit the correct autoresizing behavior. The reason is that Interface Builder disables autosizing of a parent view's children altogether if those children have no springs and struts set. To enable the autosizing behavior again, you must pass `YES` to the `setAutoresizesSubviews:` method of the parent view. Upon doing that, the child views should autosize correctly.

Design-Time Resizing Modes for Windows

Interface Builder offers two behaviors for resizing windows:

- You can resize just the window.
- You can resize the window and all of its contents.

By default, resizing a window resizes just the window. To resize the window and its contents, hold down the Command key while you click and drag the window's resize handle. Holding down the Command key causes Interface Builder to apply the autoresizing behavior currently in place for the window's contained views. This technique lets you view the runtime resizing behavior of your window or scale the window and its contents to a desired size.

You can toggle the default resizing mode by choosing Layout > Live Autoresizing from the menu. When toggled, holding down the Command key resizes just the window.

Aligning Objects

Interface Builder provides numerous tools to help you position objects precisely in your view. The following sections discuss these tools and how to use them.

About Alignment Guides

Interface Builder provides several different types of alignment guides to help you position content in your windows and views. There are three basic types of guides: automatic, dynamic, and custom.

Automatic guides appear and disappear automatically as you drag views around the design surface. Automatic guides reflect the spacing needed to lay out items according to the appropriate interface guidelines for the platform. The guides take into account the size of the controls involved as well as the guidelines for inter-view spacing. During dragging, Interface builder snaps your view to the nearest automatic guide to simplify placement. You can always tweak the placement of your view later by nudging it, as described in "Moving Views" (page 90). You can enable or disable automatic guides using the Snap To Guides menu item in the Layout menu.

Dynamic guides provide details about the position of a view relative to the edge of the window or another view. You must hold down the Option key to display dynamic guides. The exact set of guides that appear is based on the current selection (if any) and the position of the mouse. For more detailed information about using dynamic guides, see "Getting Dynamic Layout Information" (page 98).

Custom guides are guides that you place on the design surface yourself. You can use custom guides to align multiple views along some arbitrary boundary in your window and you can create both horizontal and vertical guides. Custom guides are the only guides that are saved with your nib file's design-time information. For more information about using them, see "Using Custom Guides" (page 100).

Layout information for Mac OS X applications is available in *Apple Human Interface Guidelines*. Layout information for iOS applications is available in *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines*.

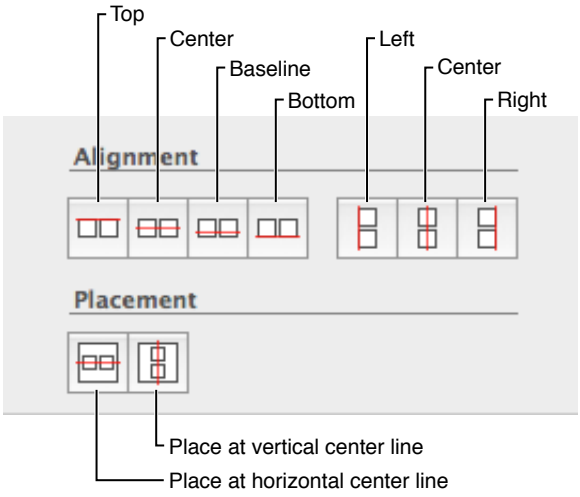
Aligning Objects Relative to One Another

To align controls relative to each other, or relative to the center of the window, select the items you want to align and do one of the following:

- Choose the desired alignment from the Layout > Alignment menu.
- Choose the desired alignment command from the size inspector.

Figure 5-6 shows the buttons in the size inspector that you use to align views. (These buttons reflect the same set of commands available in the menus.) Most of the alignment controls require you to select at least two views but the group placement controls can be used with only one selected view. The vertical and horizontal alignment controls use the layout rectangles of the selected views to perform their layout. The group placement controls center the selected views as a unit along the horizontal or vertical centerline of the window.

Figure 5-6 Alignment controls in the Size pane of the inspector



The baseline vertical alignment control lets you align controls along the baseline of any text in the control. If a control does not contain any text, Interface Builder uses the bottom edge of the control in place of the baseline. If a control has multiple baselines, Interface Builder uses the first baseline returned by the control.

Displaying Bounding and Layout Rectangles

The appearance of a view onscreen is not always an accurate reflection of its true size. Many views incorporate extra space in their frame rectangle so that they can draw shadow effects and other special adornments. The apparent size difference between the view's content and its actual frame can cause problems during layout. Specifically, if you were to lay out views based on their frame rectangles, the results might not look very good. Interface Builder compensates for this difference by letting views specify their preferred layout rectangle.

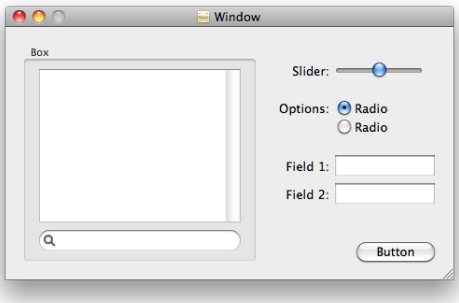
As its name implies, the layout rectangle of a view defines the bounding box to use when aligning the view with other views. Interface Builder uses layout rectangles to determine the appropriate spacing between views and to decide when to display layout guides. Interface Builder also provides a way to view both the layout rectangle and the bounding rectangle of your views:

- To display the bounding rectangle for all views, choose Layout > Show Bounds Rectangles.
- To display the layout rectangles for all views, choose Layout > Show Layout Rectangles.

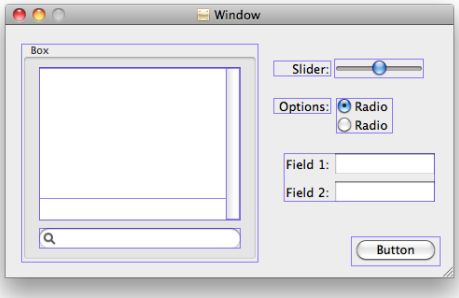
Important: Seeing the bounding rectangle for a view can help you avoid overlapping views in your design. However, you should always do your layout based on layout rectangles.

Interface Builder draws bounding rectangles in blue and layout rectangles in red as shown in Figure 5-7. You can display one or both sets of rectangles at any given time or display neither of them. You can use the information provided by these rectangles to assist you during layout.

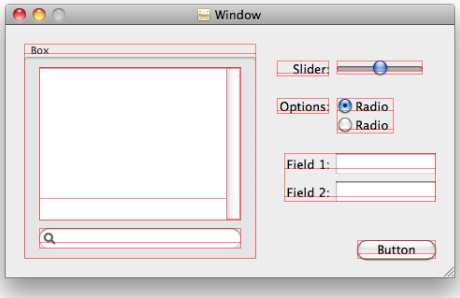
Figure 5-7 Bounding and layout rectangles for a window



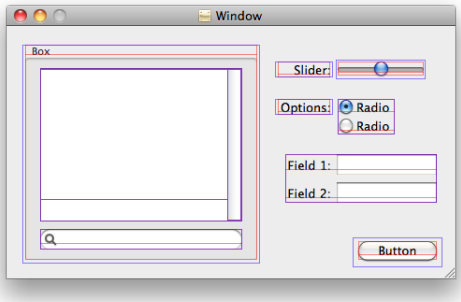
No rectangles



Bounding rectangles



Layout rectangles



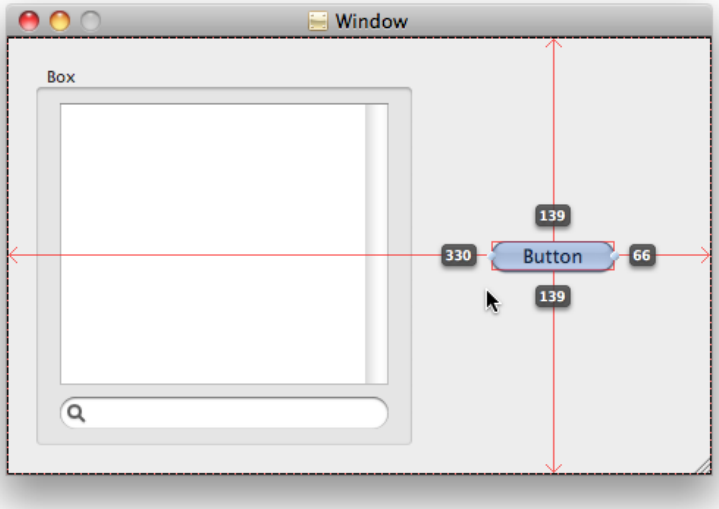
Bounding and layout rectangles

Getting Dynamic Layout Information

Interface Builder displays dynamic layout information on the design surface through the use of dynamic guides. When you press and hold the Option key, Interface Builder displays different types of dynamic guides depending on what is selected or being dragged:

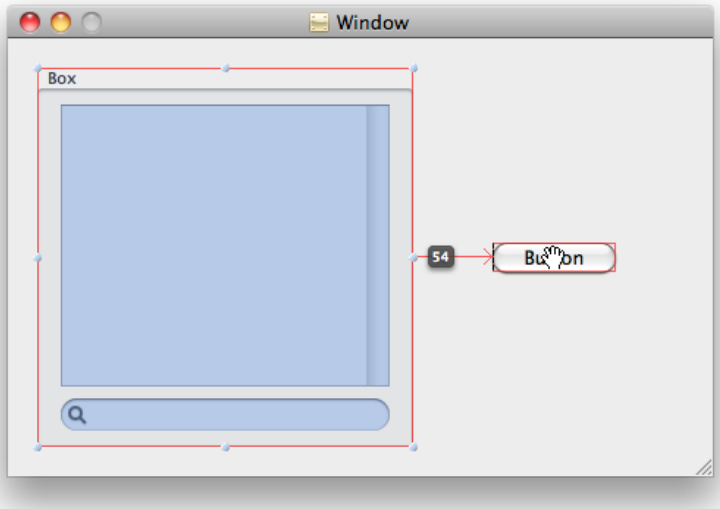
- To display guides that show distance from a view's layout rectangle to the edge of the window's content view, select the view, press the Option key, and move the mouse over the content view. Figure 5-8 shows a selected button and the dynamic guides that appear.

Figure 5-8 Option dragging a control



- To display guides that show the distance from the currently selected view to any other view, press the Option key and move the mouse over the other view. Figure 5-9 shows a window with a selected box and the distance from that box to a button in the window.

Figure 5-9 Showing the distance between two views



You can use the Option key in combination with most other modifier keys and keyboard shortcuts to display layout information while you modify the views. For example, holding down the Option key while using the arrow keys lets you see your layout information while moving a view pixel by pixel.

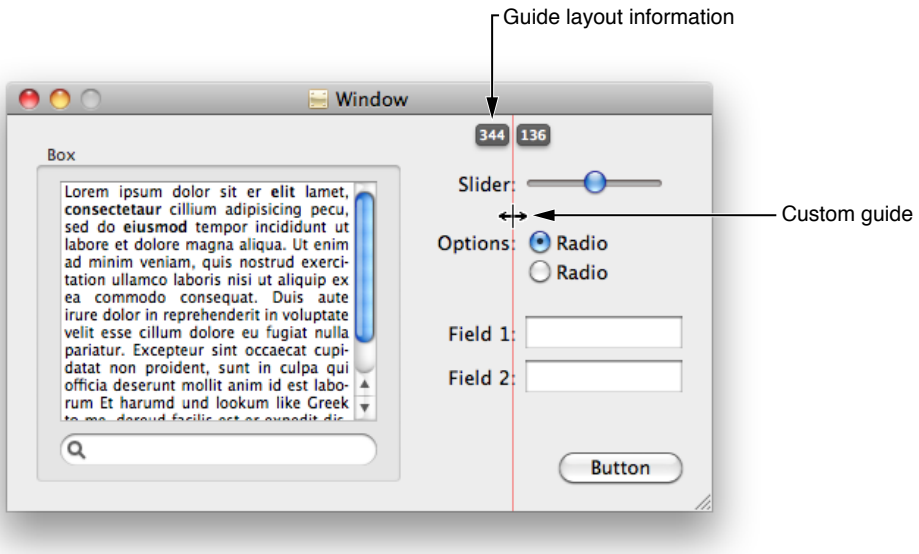
Using Custom Guides

In addition to the layout guides that Interface Builder displays automatically, you can add custom horizontal and vertical guides to a window and use them during layout. Custom guides are a useful tool for aligning groups of controls whose layout rectangles vary. You can also use them in situations where the default layout guides do not provide the alignment you want for your interface.

To add a guide to the frontmost window, choose Layout > Add Horizontal Guide or Layout > Add Vertical Guide from the menu.

By default, custom guides are placed at the center of the window to which they are added. Moving the mouse over the guide changes the cursor to either a horizontal or vertical resize cursor. When the cursor appears, you can click the guide and drag it anywhere in your window to position it. As you drag, the guide displays its distance from the relevant edges of the content view to give you more precision over its placement (Figure 5-10). To remove a guide, drag it off of the window's content view area.

Figure 5-10 Custom layout guides

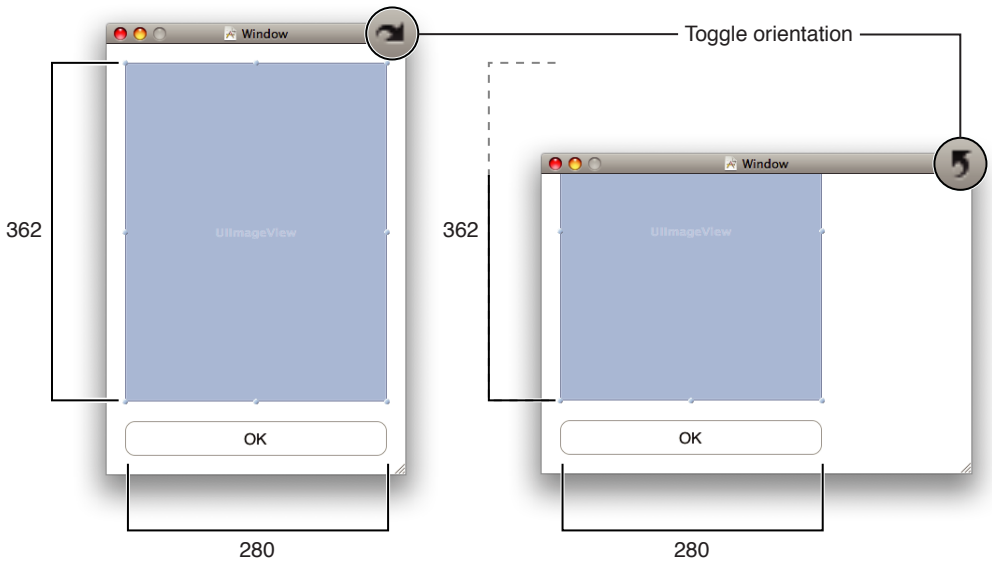


Changing the Window Orientation in iOS

In iOS, applications can support multiple orientations for their window content. Most applications support either portrait-mode orientation or a landscape-mode orientation, but applications can support both if they want. When designing your interface in Interface Builder, however, you must target your design for the orientation you plan to use when your nib file is first loaded.

To make it easier to design in either portrait or landscape mode, window objects in iOS-based nib files include a control in the upper right corner for toggling the window orientation. Interface Builder creates all windows in portrait mode by default, but clicking the control rotates the window to landscape mode. Clicking it again rotates the window back to orientation mode. Figure 5-11 shows the same window in both portrait and landscape orientations.

Figure 5-11 Changing the orientation of an iPhone window



When you change the orientation of your window, Interface Builder does not attempt to resize your views to fit the new orientation. Because nib files reflect the state of your user interface at load time, you must choose one orientation up front and design for that orientation. As a result, you should always set your window orientation prior to adding any views.

Managing Parent-Child View Relationships

Interface Builder imposes a flexible relationship between parent and child views at design time. When you first drag a view out of the Library window, it has no parent. As soon as you drop it onto your window, however, it obtains one—the view that receives the drop. From that point on, the behavior of the view is affected by its parent view, just as it would be at runtime. In particular, the following attributes of the view are affected by its parent:

- Its visible rectangle
- Its position within the window
- Its autosizing behavior

Whenever you move a parent view, its child views move with it, always staying in the same position relative to the parent view's frame. Similarly, if you delete a view, or cut or copy it, that view's children are deleted, cut, or copied as well.

Although these relationships are set when you drop the view, you can still change the relationship of a child view to its parent if needed. The following sections describe the different ways to modify the parent of a view.

Embedding Views in a Parent Container

Although you can drag views into a parent container, an easier way to configure some views is using Layout > Embed Objects In menu. This menu provides options for embedding the currently selected views in one of several standard system container views. In Mac OS X, container views include the box view, scroll view, split view, and tab view. You can also embed a view inside a custom view for which you provide the implementation. In iOS, the only supported container view is the standard `UIView` class, which you can customize or use as is.

Note: In Mac OS X, you can also use the Layout > Embed Objects In menu to embed menu items in a new submenu.

When you embed objects in another view, Interface Builder inserts that view between the selected views and their current parent. All views in the selection must have the same parent object to use this command. The new container view is made as small as possible while still containing the selected views.

If you use the Layout > Embed Objects In > Split View command to create a new split view in a Mac OS X window, Interface Builder uses the relative positions of the two views to determine the orientation of the split view. Thus, if two views are side-by-side, Interface Builder creates the split view with a vertical split bar. If one view is positioned above the other view, Interface Builder creates the split view with a horizontal split bar. In both cases, Interface Builder may reposition one or both of the embedded views so that they are aligned.

Changing a View's Parent to Another View

To change the parent of an existing view, simply drag the view to the new parent view. Dragging a view temporarily disassociates it from its parent, while dropping the view reattaches it to the new underlying view.

If you are not sure of the parent-child relationships in one of your windows, you can change the view mode of your Interface Builder document to outline or browser mode. Switching away from icon mode lets you navigate your window and view hierarchies and examine the relationships they contain. In outline mode, you can also rearrange the parent-child relationships of your objects by dragging them around the Interface Builder document window.

Using ibtool to Gather Layout Metrics

The `ibtool` command-line program is capable of outputting large amounts of detail for each object in your nib file. One of the options you can specify is the `--export` option, which writes out information about the nib file's objects to a property list file.

To generate this information:

1. Create a property list that specifies the objects and properties of interest. For more information, see the `ibtool` manual page.

2. Use a command similar to the following:

```
ibtool --export mynib.plist MyNib.nib
```

Exporting nib-file information is useful if you want to view that information or use it as a template for other interfaces. For example, a Carbon developer could use the output of a nib file to gather metrics for laying out windows and controls programmatically. Because the exported data is XML, you can parse it using the XML support found in Cocoa and Core Foundation. You can also open this file using the Property List Editor application (located in `<Xcode>/Applications/Utilities`).

If after exporting your nib object information you want to reimport some changes, you can do so with the `--import` option in `ibtool`. For example, to reimport the modified property list from the previous example, you would use the following command:

```
ibtool --import mynib.plist --write MyNewNib.nib MyNib.nib
```

When modifying exported properties, you should be aware that setting a property might alter other properties in the nib file upon reimport. During reimport, you should always use the `--write` option to output the changes to a copy of your nib file.

Tips for Creating Effective Layouts

Apple provides human interface guidelines for both Mac OS X and iOS that describe the preferred layout approaches to use when designing your user interfaces. These guidelines are there to help you create user interfaces that are both effective for users and consistent with the overall look and feel of the underlying platform.

The following sections summarize some key guidelines for layout-related scenarios that you are likely to encounter as you use Interface Builder. For information and guidance about laying out the interface of a Mac OS X application, see *Apple Human Interface Guidelines*. For information and guidance about interfaces for iOS applications, see *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines*.

Follow the Automatic Guides

The automatic layout guides provided by Interface Builder stress proper spacing and alignment for views and their associated content. As you drag views around a design surface, Interface Builder displays these guides to help you position views according to the human interface guidelines for the target platform. Although Interface Builder does not force you to place views according to the guides, it is always a good idea to pay attention to them and use them whenever possible.

Use Proper Alignment for Labels and Controls in Mac OS X

In Mac OS X, windows use a center-equalized layout, whereby controls are aligned with the vertical center of the window. If a control has an associated label, that label should appear to the left of the control and have right-aligned text. The control's text should be left aligned to create the centered appearance.

If you have groups of controls in a vertical column in your window, it is also important to make the width of those controls the same whenever possible. Consistent sizing of the control bounding boxes creates cleaner lines and is generally more visually appealing.

Group Related Controls in Mac OS X

In Mac OS X, an effective interface design uses visual cues to highlight the relationship of different views and controls. Views that are located close together have a relatively strong association with one another. Views that are separated from each other by some sort of divider have a weaker association. You can use these visual associations to convey information about the purpose of different views and their relative importance to the user's current task.

Mac OS X windows have several different options for creating visual associations. Table 5-2 lists these options and the associations they create along with any limitations they impose on layout. For more information about organizing the visual content of your user interface, see *Apple Human Interface Guidelines*.

Table 5-2 Techniques for grouping controls

Group delimiter	Notes
Boxes	Provides the strongest group association but also requires the most space within a window.
Separator line	Provides strong group associations when space is at a premium.
Whitespace	Provides good group associations when space is less of an issue. Typically, you would use this technique for smaller groups of controls.

Object Attributes

One of the biggest advantages of using nib files in an application is that when you load that nib file, the objects inside retain the same attribute values that you set in Interface Builder. Rather than have to reconfigure your objects programmatically, the nib-file objects are ready to go almost immediately after being loaded into your program.

You configure most object attributes using the inspector window. The attributes and identity inspectors expose the more commonly configured attributes for an object. Other panes may also expose important attributes for configuring the visual style of a view. There are a handful of views that support direct editing of text-related attributes on the design surface. Finally, some text-oriented views also support the use of a formatter object, which defines rules for presenting the text in that view.

The following sections describe the different techniques for modifying object attributes in Interface Builder. These sections do not cover the specific attributes of the objects themselves. Object attributes are tied to the exposed attributes of the underlying class, and therefore the class reference documentation and human interface guidelines for the target platform are the definitive source for that information.

Editing Object Attributes

There are several ways to modify an object's basic attributes:

- Select the object and change the desired value in the inspector window.
- Select the object and change the desired font or color using the font or color panel. (Not available for all views.)

  To display the font panel, choose Font > Show Fonts. To display the color panel, choose Font > Show Colors. For example, to change the background color of a table view, select the view, bring the color panel forward, and choose the desired color.
- Double-click the object in the design surface to edit relevant attributes. (Not available for all views.)

  If an object displays any text, double-clicking the text string usually opens a field editor that you can use to change the text. For controls such as checkboxes that display a selection state, clicking the control lets you set the default state. Double-clicking a control with a menu (such as a pop-up button) opens the menu for editing.

Changes you make in the inspector window take effect immediately. To revert a change, use the Undo command or change the value in the inspector again.

If multiple objects are selected, the inspector window lets you change as many attributes as makes sense for the selection. If all of the selected objects have the exact same underlying class, the inspector shows you all of the same information you would get if just one object were selected. If the classes differ, the inspector shows you all of the attributes that the selected objects have in common. For example, selecting a push button and a text field shows only the control and view attributes in the Attributes pane, while other panes display reduced sets of information.

Attaching Graphical Effects in Mac OS X

In Mac OS X v10.5 and later, you can associate graphical effects with your views using the Effects pane of the inspector window. Graphical effects change the visual appearance of a view. For example, you can use the Effects pane to make a view partially transparent or to apply a Core Image filter to its content. You can also assign transition behaviors to the view to change the way subviews appear and disappear when they are added to or removed from the view.

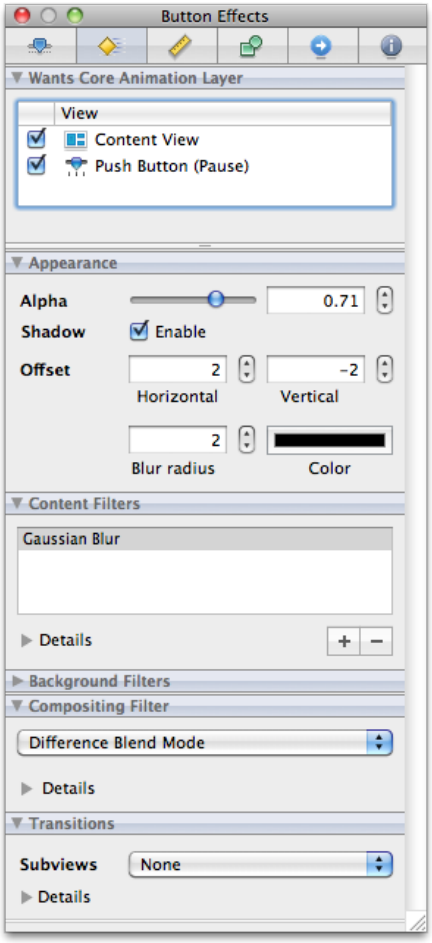
Most of the animation and filter effects behaviors are implemented using Core Animation. Core Animation is a lightweight drawing layer that combines the power of Quartz, Core Image, and OpenGL with the simplicity of Cocoa views. Core Animation takes advantage of the available graphics hardware to accelerate rendering operations and make it possible to apply complex visual effects in real time.

For more information about Core Animation effects and how you use them in your Cocoa applications, see *Core Animation Programming Guide*.

The Effects Inspector

You use the effects inspector (Figure 6-1) to apply Core Animation effects to your views. The inspector is organized into several subsections containing different types of visual effects.

Figure 6-1 Effects inspector



The Effects pane of the inspector window is divided into several sections, each of which lets you add specific visual effects to your view. Table 6-1 summarizes the sections available in the inspector and the visual effects each one applies.

Table 6-1 Supported effects

Section	Description
Wants Core Animation Layer	Lists the selected view and its parent views. Clicking the check boxes in this section enables layers for the given view and any child views.
Appearance	Lists the basic visual effects you can apply to your view. You use this section to apply transparency and shadow effects to your view. Shadow effects that fall outside the bounds of your view are clipped by non Core Animation superviews.
Content Filters	Applies Core Image effects to the selected view and its subviews.
Background Filters	Applies Core Image effects to the content located behind the view. These filters are applied to any background content prior to compositing that content with your view.

Section	Description
Compositing Filter	Applies a Core Image filter that specifies how your view should be composited with a background image or with the content behind your view. Compositing lets you blend your view contents with other views or images.
Transitions	Adds transition effects to the selected view. These effects are applied to subviews whenever those subviews are added to or removed from the view. Transitions represent an expanded version of the behavior offered by the `NSViewAnimation` class.

For information about how to apply the Core Animation effects associated with your views in your nib file, see *Core Animation Programming Guide*.

Enabling Graphical Effects for a View

To enable graphical effects for a view, click the check box for that view in the Rendering Tree section of the effects inspector. This section lists the currently selected view and its superviews. Graphical effects require the presence of a Core Animation layer. Enabling a view's check box adds such a layer to that view and all of its child views and is equivalent to setting the `wantsLayer` property of the view to `YES`.

When you enable layers for a view, Interface Builder displays some of the graphical effects you configure in the effects inspector, but may not display them all. Most effects involve input from your application that may not be available in Interface Builder. For example, transitions are not shown in Interface Builder because they are triggered only when you add or remove subviews at runtime.

Configuring Filters

Filters let you apply interesting visual effects to the contents of your views. You can use filters to create a specific look for your user interface, to highlight specific portions of your interface, or to apply visual effects to user-defined or dynamically changing content. The advantage of using filters is that you do not have to render the content in advance. You simply add the desired filter to your view and Core Animation composites the filter along with the rest of your view's content. This allows you to create and update your content in real time while still providing stunning visual effects.

You can use the effects inspector to configure three basic types of filter for your views:

- Content filters
- Background filters
- Compositing filters

A view can have only one compositing filter but may have multiple content and background filters. To enable a compositing filter, select the desired filter from the list and configure its options. To configure content and background filters, do the following:

1. In either of the Content Filters or Background Filters sections, click the plus (+) button to add a new filter. A filter selection sheet appears.

2. From the filter selection sheet, select the filter you want to apply and click OK. The filter should appear in the corresponding filter table of the section.
3. To configure the new filter, select it and expand the Details section below the table to display the filter attributes.
4. Change the filter attributes as desired.

You can apply multiple content and background filters to a single view. The filters you add are applied in the order listed in the content filters or background filters tables. To delete a filter, select it from the appropriate table and click the minus (-) sign.

Configuring View Transitions

View transitions are a way to provide the user with visual feedback whenever you change your view hierarchy by adding or removing views. View transitions make adding and removing views less jarring for the user. For example, removing a view whose parent has an attached fade transition causes the view's opacity to decrease gradually to 0, at which point the view is actually removed from the view hierarchy. The transition provides a visual cue to the user that the view has been removed.

In the effects inspector, you specify the transition type to use whenever subviews are added to or removed from the selected view. Transitions are not inherited by child views. They apply only to the immediate children of the selected view.

To activate a transition at runtime, simply add or remove subviews from the parent view. The Core Animation framework automatically applies the associated transition to the affected views without further prompting from your code. View transitions themselves are simply Core Image transition effects associated with the Core Animation layer of a view. Core Image provides a set of default transitions for you to use and you can extend this set to include custom transitions you create.

Note: Core Animation effects are available for applications that run in Mac OS X v10.5 and later. If you want to include view transitions for Mac OS X v10.4 as well, use the `NSViewAnimation` class instead. For more information, see *Cocoa Drawing Guide*.

For information on how to implement Core Image transitions, see *Core Image Programming Guide*.

Applying Formatters

Formatters are special objects that modify the textual representations of Cocoa objects. Formatters provide a fast and easy way to format complex text information without writing any code. For example, you might use a formatter in a text field to format a sequence of numbers as a currency or date value. For controls that can accept text input from users, formatters not only reformat the entered text, they also validate it to ensure it does not contain any illicit characters.

Cocoa defines two concrete formatter classes that you can use as-is in your interfaces. The `NSNumberFormatter` class lets you format strings containing integers, floating-point values, currency values, percentage values, and so on. The `NSDateFormatter` class lets you format strings containing date and time information. You do not have to write any code to use these objects. Instead, each object supports a flexible

notation that you enter in the inspector to specify how you want text formatted. When subsequently attached to an object, the formatter automatically formats the corresponding object's contents based on the parameters you specified.

If you want to format something other than number or date values, you can create your own custom formatters and integrate them into Interface Builder using plug-ins. The `NSFormatter` class provides the basic interface used by formatters to interact with Cocoa controls and cells. For information on how to create a custom formatter, see *Data Formatting Guide*.

Note: If you are an iOS developer, you can use formatter objects to format text strings prior to display. Formatter objects must be applied from your code, however, and cannot be configured in Interface Builder. Carbon developers must similarly use formatters from their code but must use Core Foundation formatters instead of the Cocoa formatter objects. For information about using formatters programmatically, see *Data Formatting Guide* or *Data Formatting Guide for Core Foundation*.

Attaching a Formatter to an Object

In Cocoa applications, there are two ways to attach a formatter to an object:

- Drag the formatter object from the Library window and drop it on the control. (Recommended)
- Drag the formatter object to the top level of the nib document window and connect it to the `formatter` outlet of the target control.

When you drag a formatter object around your window, Interface Builder highlights any controls that can accept that formatter. Essentially, you can attach a formatter to any Cocoa object whose cell defines a `formatter` outlet. When you drop a formatter onto an object, Interface Builder attaches the formatter to the cell but does not mark its `formatter` outlet as connected. Instead, it configures the outlet lazily at runtime when the nib file is loaded. To show that the object does have a formatter, Interface Builder displays a badge next to the object when it is selected. To select the formatter, click its badge.

Dropping a formatter on a control associates the formatter with that control, configuring all of the necessary connections automatically. From that point, all you have to do is configure the formatter parameters in the inspector and you are done. In addition to being easier to configure initially, this technique reduces the overhead required to manage the formatter object at runtime. Formatters attached directly to an object are released automatically when the object itself is released. Dragging a formatter object to the nib document window adds that formatter to the top level of your nib file. Objects at the top level of the nib file are not released automatically and must be released explicitly by the code that loaded them.

The only time you might want to place a formatter object at the top level of your nib file is in cases where several objects need to share the same formatter object. Objects that share a formatter also share the same formatting behavior, and any changes you make to the formatter affect all connected objects. For information on how to connect objects to outlets, see "Creating and Managing Outlet and Action Connections" (page 120).

Note: You should never attach more than one formatter to an object in your nib file.

To remove a formatter that you dragged to an object, select its badge and choose Edit > Delete or press the Delete key.

Configuring Number Formats

After adding a formatter to a Cocoa object, you should select that formatter and configure its settings. The default Cocoa formatters provide a flexible notation for specifying how strings are formatted at runtime. For number formatters, you use this notation to specify the number and type of digits to use, whether to include special symbols, and any potential limits on the number's value.

The `NSNumberFormatter` class supports two syntaxes for formatting strings:

- In Mac OS X v10.4 and later, formatters support the conventions defined in Unicode Technical Standard #35. (Recommended)
- In all versions of Mac OS X, formatters support a set of custom conventions defined in Number Format String Syntax (Mac OS X Versions 10.0 to 10.3) in *Data Formatting Guide*.

The Unicode conventions include significant enhancements over the older conventions and are recommended for use in any new applications you create. "Collection Views" lists the key field groups in the inspector and how you use them to configure your format strings.

Table 6-2 Inspector fields for modern formatters

Para	Para
Style	Default styles for standard types of numerical formatting. Selecting a value from the pop-up menu populates other inspector fields with reasonable starting values.
Format	The core format strings for positive and negative values. These fields reflect the current format string derived from the other inspector options. You can also use these fields to modify the format string directly.
Padding	The padding to apply to the prefix or suffix portion of the format. Each string format can have a prefix, a numeric part, and a suffix. The prefix and suffix portions usually correspond to things like currency or percent symbols that are integral to the format but not part of the numerical portion. You can use the controls associated with this field to specify how many padding characters (and which character) to include before or after the prefix or suffix. Padding may be inserted at only one of the specified locations. Changes you make to this field are reflected in the Format fields.
Rounding and Increment	The rounding rules to apply to the number. You can round numbers up or down in various ways and can specify the rounding increment to use. For example, using the half-even setting with an increment of 50, the value 1230 would round to 1250. Changes you make to this field are reflected in the Format fields.
Multiplier	The factor to use when converting between numerical values and the corresponding strings. Multipliers allow the formatted value to be different than the value used internally by your program. For more information, see *NSNumberFormatter Class Reference*.
Grouping	The number of digits to group together between separator characters. The primary grouping represents the digits immediately to the left of the decimal point. Secondary groupings represent the spacing for the remaining digits to the left of the first separator. Specify 0 for the secondary grouping value to use the primary grouping throughout. You can also use the grouping checkboxes to configure additional grouping-related options. Changes you make to this field are reflected in the Format fields.

Para	Para
Constraints	The constraints to apply to the resulting number. You can set limits on the value itself or on the number of digits it contains. You can also specify the default symbols to use, although typically these are obtained from the current locale.

For additional information and examples on how to specify formatting strings using the older notation, see *Data Formatting Guide*. For more information on formatting strings using the Unicode conventions, see Unicode Technical Standard #35.

Configuring Date Formats

After adding a formatter to an object, you should select that formatter and configure its settings. The default Cocoa formatters provide easy-to-use tools for configuring format options from a collection of tokens. If you prefer to use the advanced notation defined in Unicode Technical Standard #35, an advanced editor is also available for configuring your formatter using a specially-formatted string.

For each date formatter object, you must specify the versions of Mac OS X your nib file must support. The implementation of the `NSDateFormatter` object has improved over time to support more advanced features. Most nib files should support the conventions from Mac OS X v10.4 and later, but if you need to support earlier versions of the system that option is available.

For additional information and examples of how to specify formatting strings, see *Data Formatting Guide*.

Setting the Key Equivalent for a Button

In Carbon and Cocoa nib files, you can associate key equivalents with a button to facilitate pressing that button more quickly. Typically, this feature is used to associate the Return or Escape key with the button to indicate its status as the default button or the designated cancel button of a window. In Cocoa nib files, you can assign other keys to buttons to create shortcuts as well.

To assign a key equivalent for a Cocoa button, do the following:

1. Select the button.
2. Open the attributes inspector.
3. Click the Key Equiv. field so that it is highlighted.
4. On the keyboard, press the key (or key combination) you want to assign to that button.

   You can assign key combinations by pressing any combination of the Command, Control, Option, or Shift modifier keys in addition to the desired key.

Carbon nib files support the configuration of the default and cancel buttons on a window but do not support arbitrary key assignments. The default button is triggered when the Return key is pressed and the cancel button is triggered when the Escape key is pressed. To assign either of these equivalents to a button, open the attributes inspector for the button and select the appropriate value from the Type field.

Configuring the Tab Order for Views

Tabbing between views is a common way for the user to move quickly between a set of text fields in a window, enabling the fast entry of data using only the keyboard. When the user presses the tab key, focus moves from view to view, usually in a circular pattern. For Carbon developers, the code to manage this change in keyboard focus must be written manually in Xcode. For Cocoa developers, however, Interface Builder lets you configure the tab order using outlet connections in the `NSView` class.

When the user presses the tab key, Cocoa uses the `nextKeyView` outlet of the view that currently has the focus to determine where the focus should move to next. To set up the tab order for a particular window, all you have to do is configure this outlet for each view. It is recommended that you configure all of your views at once, moving from one view to the next to ensure that the chain is really circular and comes back to the right starting point.

To specify the view that should receive the focus when the window is first shown, connect the `initialFirstResponder` outlet of the window to the desired view.

For more information about configuring outlet connections, see "Creating and Managing Outlet and Action Connections" (page 120).

Configuring User Assistance Attributes

Providing users with information about your interface can significantly improve the user experience. From Interface Builder, you can configure two different types of help information:

- Help tags (also known as tool tips)
- Accessibility information

Help tags are short strings that appear when the user hovers the mouse over a control in Mac OS X applications. You can use help tags to provide the user with basic information about the purpose of a view or give some indication of how it should be used. You specify the help tag for a given control using the Identity pane of the inspector.

- For Cocoa nib files, use the Tool Tip section of the identity inspector to set the help tag text.
- For Carbon nib files, use the fields in the Object Help Identity section of the identity inspector to set help tags

For Carbon controls you can specify both a simple help tag and an extended help description. You use the Help field to specify the basic help text that appears when the user hovers the mouse over a view. You use the Extended Help field to provide an extended version of the help text that appears when the user presses the Command key. Cocoa controls support only the basic help tag information. Nib files in iOS do not support help tags.

Note: For guidelines on how to write your tooltips, see the section on User Assistance in "Using Mac OS X Technologies" in *Apple Human Interface Guidelines*.

For Cocoa nib files, another way you can provide assistance is by adding Section 508 information (also known as accessibility information) to your controls and views. System technologies such as VoiceOver use accessibility information to describe user interface features to users with impaired vision or other disabilities. Accessibility support is an important feature for many industries and is mandatory if you sell your software into the government sector.

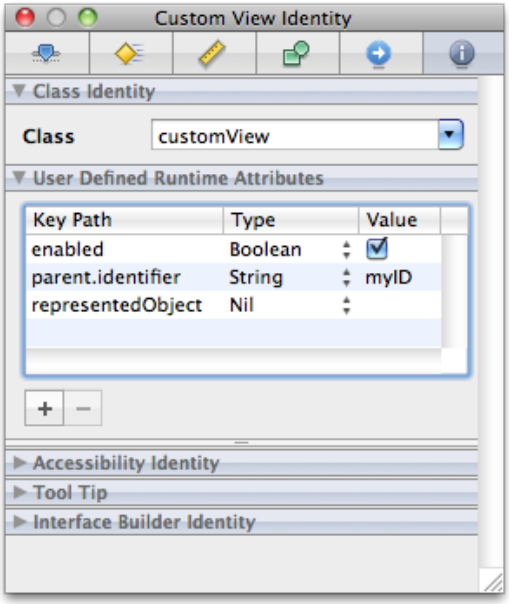
You specify accessibility text using the Identity pane of the inspector window. Use the Description field to provide the user with information about an object's purpose. You can also use the Help string to provide a short hint on how to handle the object, although Cocoa uses the view's help tag by default if this field is empty. For more information about accessibility and Cocoa, see *Accessibility Programming Guidelines for Cocoa*.

Note: For Carbon nibs, you must add accessibility information programmatically. For information on how to do so, see *Accessibility Programming Guidelines for Carbon*

Configuring Runtime Attributes for Custom Objects

This feature is useful for a custom object that doesn't have its own Interface Builder inspector. You can configure the custom object to initialize a list of attributes when the nib file is loaded. You specify the list in a table in the identity inspector called "User Defined Runtime Attributes". Each entry in this table results in `setValue:forKeyPath:` being called against the inspected object when the nib file is loaded.

For example, if you add the following entries in the identity inspector for a custom view:



The custom view will get these messages when the nib is loaded:

```
[customView setValue:[NSNumber numberWithBoolean:YES] forKeyPath:@"enabled"];
[customView setValue:@"myID" forKeyPath:@"parent.identifier"];
[customView setValue:nil forKeyPath:@"representedObject"];
```

Specifying Carbon Attributes

You can use Interface Builder to create nib files containing HIToolbox types for your Carbon applications. Carbon nibs support the inclusion of window and menu resources only, which simplifies the nib creation process somewhat. Rather than create connections between objects at design time, you use Interface Builder to assemble the visual appearance of your windows and menus and then implement all of the behavior for those objects in your code. For information on how to attach controller code to your user interface, see *Handling Carbon Windows and Controls*.

Assigning Command IDs to Objects

Command IDs correspond to the high-level actions an application should take. You can associate command IDs with the controls and menu items in your Carbon nib file to indicate what action should be taken when that control or menu item is used. Whereas a Control ID is unique for a given control, the same command ID can be assigned to multiple objects. For example, you could could associate the Zoom command with a Zoom menu item, a Zoom button in your document window, and the Command-Z keyboard equivalent.

You associate a command ID with a particular control or menu item using the inspector window. Carbon defines several standard command IDs that you can reuse, or you can define custom codes to represent the actions the user can take in your application. To assign a Command ID to a control or menu item, do the following:

1. Select the control or menu item.
2. Open the inspector window and select the Attributes pane.
3. From the command pop-up menu, select the command you want to associate with the object. You can use one of the predefined IDs provided by Carbon or choose `<other>` to enter the four-character corresponding to your custom command.

For more information about command IDs and how they are used, see *Carbon Event Manager Programming Guide*.

Assigning a Signature and Control ID

To access controls from your application, you need to assign each control a signature and control ID. The signature is a four-character code that typically is the same as the application signature. The control ID is an integer you use to identify the control. The combination of these values uniquely identify the control to your application..

To assign a signature and control ID:

1. In your nib file, select the control.

2. Open the inspector window and select the Attributes pane.
3. In the View section, set values for the Control Signature and ID fields.

For more information about working with Carbon controls, see *Handling Carbon Windows and Controls*.

Adding Auxiliary Properties

The inspectors for Carbon controls and HIView objects provide tables where you can associate custom properties with your controls and views. Custom properties are application-defined pieces of data that your code uses to configure the control at runtime. For example, if you have a text field control, you might include a property that signifies that the control supports only numerical values. At runtime, your validation code could look for this property and use it to restrict the values entered in this control by the user.

You can add auxiliary properties to both Carbon controls and HIView objects using the Attributes pane of the inspector window. Each property consists of a creator code, tag code, type, and value. The creator and tag codes are four-byte values you use to identify the property at runtime. The creator code is typically the same as your application's four-byte creator signature. Although Carbon treats properties as generic blocks of untyped data at runtime, Interface Builder uses the type information you provide to help package the data appropriately in your nib file.

To load a property at runtime, you use the `GetControlProperty` function of the Carbon Control Manager. When calling this function, you specify the creator and tag codes for the desired property and get back a buffer containing the value for that property (if it is present). For more information about using this function to load properties, see *Control Manager Reference*.

Adding Custom HIView Parameters

If your nib files contain any custom HIView objects, you can associate custom parameters with those views in addition to any auxiliary properties. When your view is initialized at runtime, the parameters you specify in the nib file are included as parameters to the `kEventHIObjectInitialize` event. You can use these parameters to initialize your view much like you would use auxiliary properties; see "Adding Auxiliary Properties" (page 116).

HIView parameters consist of a parameter name, type, and value. The parameter name is a four-byte code that uniquely identifies the type of parameter to your application. The type and value information specify the actual data for the parameter and their contents are left for you to determine. To retrieve an event parameter at runtime, use the `GetEventParameter` function, defined in *Carbon Event Manager Reference*.

Attribute Guidelines

When building your interface, you should consider the intended use of the windows and menus in your application. Although Interface Builder helps you to create good looking user interfaces in many ways, it is still just a tool that does what you ask it to do. It is up to you to make appropriate design choices as you build your interface.

A good source of information about interface design is the human interface guidelines for the target platform:

- *Apple Human Interface Guidelines* for Mac OS X
- *iPhone Human Interface Guidelines* and *iPad Human Interface Guidelines* for iOS

All interface designers should read the human interface guidelines at least once to understand the basic design principles for the platform. The following sections provide additional guidance for you to consider as you create your windows and menus in Interface Builder.

Use Consistent Control Sizes in Mac OS X

In Mac OS X, controls support up to three different sizes: regular (or large size), small, and mini. Some controls support all three sizes, and some support only the regular and small sizes. Although choice is great, mixing controls of different sizes in the same window is not. Mixing different sized controls looks strange and makes aligning those controls more difficult.

Rather than use controls of different sizes, you should always choose a size that is appropriate for your window and stick with it. Using controls of the same size ensures that the baselines and bounding rectangles used during layout match and create a consistent appearance. Although Interface Builder does not warn you when you mix control sizes, the ability to edit the properties of multiple objects simultaneously makes it easy for you to adjust the size of nonconforming controls later. For more information, see "Changing the Size of a Control" (page 94).

Assign Names to Your Objects

Although not part of most guidelines, naming your nib file objects can make it easier to edit and localize your nib files later. By default, new objects added to a nib document receive a generic name, such as "Window" or "Custom View". If your nib file contains multiple objects of the same type, it can be hard to distinguish one from another quickly. Assigning unique names to your objects rectifies this problem. Names can also help translators understand the purpose of a particular object, which can aid in making the appropriate translation.

To set the name of an object, do the following:

1. Select the object.
2. Open the identity inspector.
3. In the Interface Builder Identity section, type the desired name in the Name field.

Names you specify in the identity inspector do not appear in your actual user interface. They are simply cues that Interface Builder displays when you edit your nib file.

In addition to assigning names, you can also add notes to your objects. Notes can provide even more help to translators by offering additional context about how an object is used in your user interface. Like names, notes reside in your nib file and never appear in your user interface.

Connections and Bindings

For Mac OS X and iOS developers, the ability to connect objects at design time is a key part of the rapid development aspect of applications. Many Cocoa classes use connections to implement basic behaviors, including event handling, window management, focus management, and action dispatch. Connections are therefore an important consideration in the design of your Mac OS X Cocoa and iOS applications.

This chapter covers the techniques for creating connections and bindings using Interface Builder.

Note: The information in this chapter applies to nib files developed for Cocoa and iOS applications only. Carbon applications do not support connections. Instead, Carbon objects must be configured with command IDs that are then used by the underlying code to respond to interactions with views and controls. For more information on configuring Carbon nib files, see "Specifying Carbon Attributes" (page 115).

About Connections and Bindings

There are four fundamental types of connections you can create in a Mac OS X or iOS application:

- Outlet connections
- Action connections (Mac OS X only)
- Event connections (iOS only)
- Bindings (Mac OS X only)

An **outlet** connection is a special type of instance variable that you specify in your source files and configure using Interface Builder. You use outlets to store references to important objects in your nib file, such as important views, controls, and controller objects. Your view classes can similarly use outlets to store references to related objects. For example, many Cocoa views use outlets to locate the data formatters, delegates, and contextual menus they should use. Because they are instance variables, you can modify outlets programmatically if you wish, but doing so is unusual. In most cases, you connect an outlet to its target object in Interface Builder and do not change the outlet later.

Note: The name "outlet" comes from the notion that when you connect an object to an outlet, you "plug in" the object to that outlet.

An **action** connection is a message-passing relationship between a control and a target object in a Cocoa application. Whenever the user interacts with a control in a predetermined way or selects a menu item from a menu, the control or menu invokes the action method of its associated target object. You can use action methods to perform whatever tasks are needed to respond to the given interaction. For example, if the user clicked an Apply button in a panel, your action method could apply the specified settings to your application's data structures. Clicking a push button, checking or unchecking a checkbox, or selecting an item from a pop-up menu button are all examples of control interactions that generate action messages.

An **event** connection is similar in nature to an action connection but is specific to iOS applications. Controls in the UIKit framework can send different action messages for different types of interactions within a control. For example, a control can send separate messages when the user first touches the control and then subsequently lifts that finger from the screen. In addition each event can have multiple target objects, so that when the event occurs, different action messages are sent to each object.

Cocoa **bindings** are a way to connect your application's user interface to its underlying data without writing a lot of glue code to synchronize the two. Bindings take advantage of the key-value coding (KVC) and key-value observing (KVO) protocols to bridge the data in your data structures with the views you create in Interface Builder. Whenever the user changes a value in a view, the bindings code automatically propagates that change down to the associated data structure. Similarly, if your program changes the data by calling an accessor method, the bindings code propagates that change back to the associated view.

Creating and Managing Outlet and Action Connections

You can manage outlet, action, and event connections in Interface Builder using the inspector window or the connections panel. These two tools provide the same basic behavior but do so in slightly different ways. The Connections pane of the inspector window displays the connections associated with the currently selected object. Only the connections for the currently selected object are displayed and changing the selection changes the contents of the inspector. The connections panel, on the other hand, is an on-demand window that you can create for one or more objects simultaneously. Unlike the inspector, a connections panel displays the connections for the same object until the panel is dismissed. Its contents do not change to match the current selection.

The connection creation behavior in Interface Builder is designed to be as flexible as possible. The sections that follow describe the techniques you can use to create connections in your nib files.

Making Quick Connections By Dragging

If you want to make a single outlet or action connection quickly, you can do so directly from the design surface or your Interface Builder document window.

1. Control-click (or right-click) the source object of the connection and do not release the mouse button.
2. While holding the mouse button, drag to the target object.
3. Release the mouse button over the target. Interface Builder displays a prospective list of actions and outlets.
4. Select the desired outlet (of the source object) or the desired action or event (of the target object) to create the connection.

When you use the control-drag gesture to create a connection for an action, the final part of the drag involves picking the action name from a list of actions. Outlets have a similar list of outlets. If there are previous connections that have the same name, the menu items to pick those names will be flagged. If an identical connection to the one being made already exists, the menu item will be flagged with a dot. If a connection exists with that name, but to a different object, it will be flagged with a dash.

For action connections in Cocoa, this technique is equivalent to opening the connections panel and configuring the source object's sent actions connection. Because a source object in Cocoa can send its action message to only one target object, you should use this technique only once to configure a given source object's action connection. Repeating the process for the same source object would break the old connection before establishing the new one. You can use this technique, however, to configure each of the source object's outlets.

Making Connections Using the Connections Panel

The connections panel is a convenient way to create multiple connections for the same object in quick succession. To use the connections panel, do the following:

1. Control-click (or right-click) an object and release the mouse button to display the panel.
2. Click in the circle to the right of the action, outlet, or event you want to connect and begin dragging.
3. Drag the mouse over the target object for the connection. (If the target of a connection is not visible, hovering over its parent object causes the parent to open and reveal its children.)

   The target object should highlight to indicate a connection is possible. If it does not highlight, the target object is not of the right type and cannot be connected to the source object in that way.
4. If the target object highlights, release the mouse button to create the connection.
5. If you are using the connections panel to configure the sent action or referencing outlet for the object, Interface Builder displays a list of the target object's action methods or outlets. Select the desired action method or outlet from the list to finish the connection.

   Other types of connections do not require this configuration step.

After you establish a connection, the connections panel fills the circle next to that action or outlet and displays information about the connection. Each connection also includes a close box icon that you can use to break the connection. If a received action has multiple source objects associated with it, the panel displays a disclosure triangle, which you can use to reveal the individual connections. If you select a connection, Interface Builder displays a path control at the bottom of the panel and a 'take me there' button for the selected connection.

The connections panel remains visible as long as you use it. If you click outside the panel (by selecting a different object in your window, for example) Interface Builder dismiss the panel automatically to get it out of your way. You can also dismiss the panel explicitly by clicking its close box.

If you want to select other objects in your nib file but do not want the connections panel to disappear, simply drag the panel to a new location. (You do not have to drag the window far; even dragging it a single pixel is sufficient.) Dragging the panel lets Interface Builder know that you want to continue using it. If you do this, however, you must dismiss the window yourself by clicking its close box when you are done.

Example: Creating Action Connections

There are two ways to create connections using the connections panel:

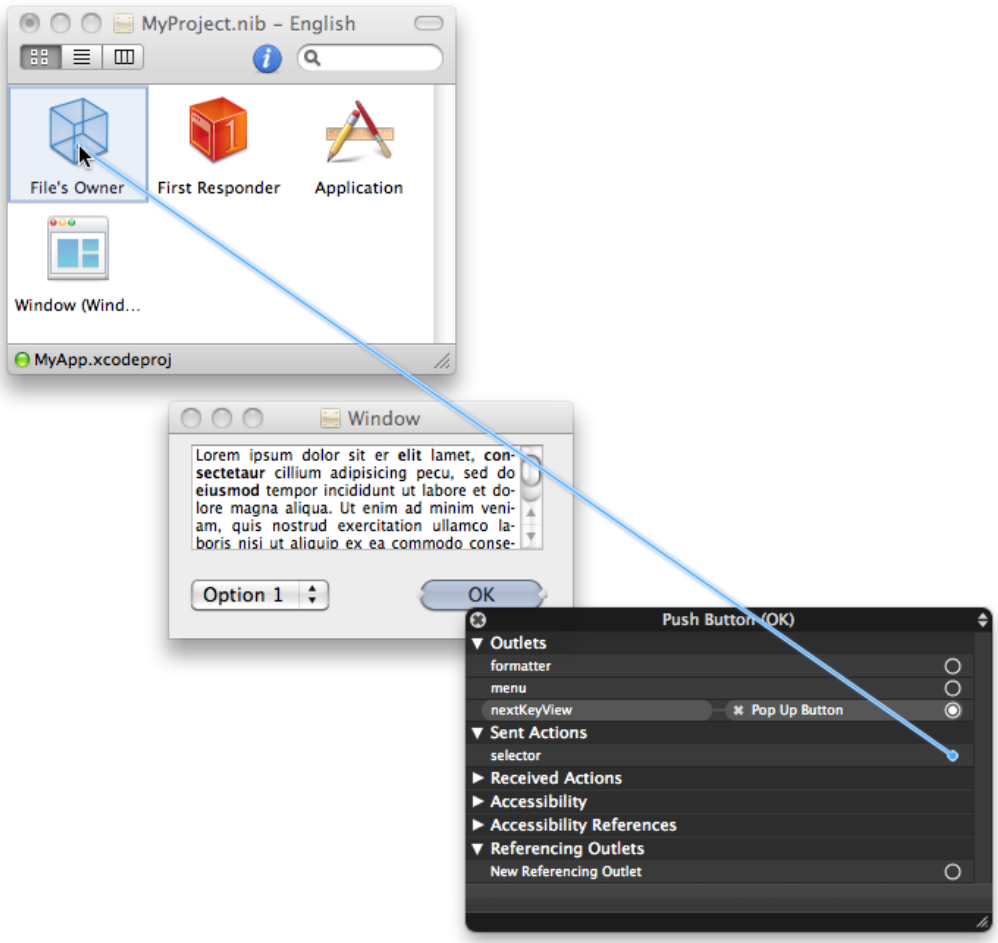
- You can start at the source object:
  - For Cocoa applications, connect from the sent action selector.

- For iOS applications, connect from one of the specified events.

- You can start at the target object and connect one of its action methods to the desired source object.

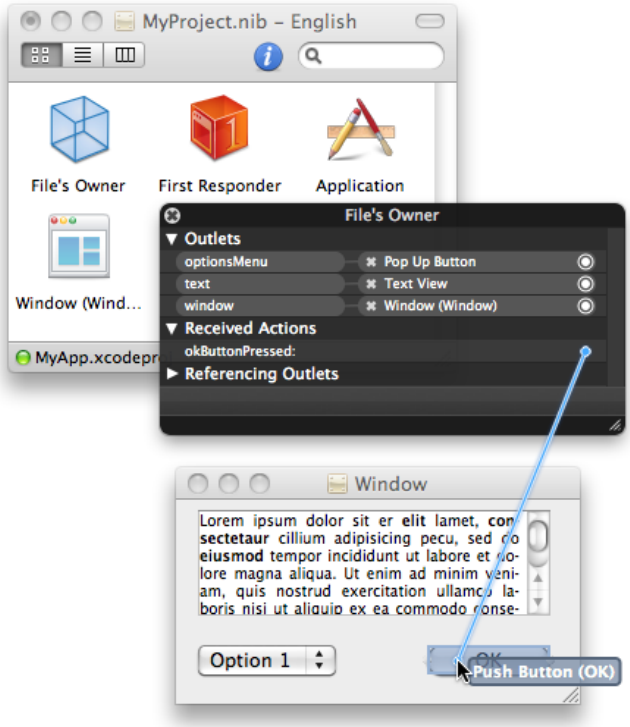
To create a connection starting at the source object, Control-click it to display the connections panel and scroll to the appropriate entry. Click in the circle next to the entry and drag to the desired target object that you want to receive the action or event message, as shown in Figure 7-1. When you let go of the mouse over a valid target object, Interface Builder displays a second connections panel that lists the action methods of the target object. Selecting a method from the list completes the connection.

Figure 7-1 Connecting from source to target



To create a connection starting at the target object (the one that defines the action method), control-click the target object to display its connections panel. Click in the circle next to the desired action method and drag to the source object, as shown in Figure 7-2. When dragging from the target to the source object in a Cocoa application, Interface Builder does not need to display a second panel at the source object. For iOS applications, however, it displays a second panel so that you can choose the event you want to trigger the action message.

Figure 7-2 Connecting from target to source

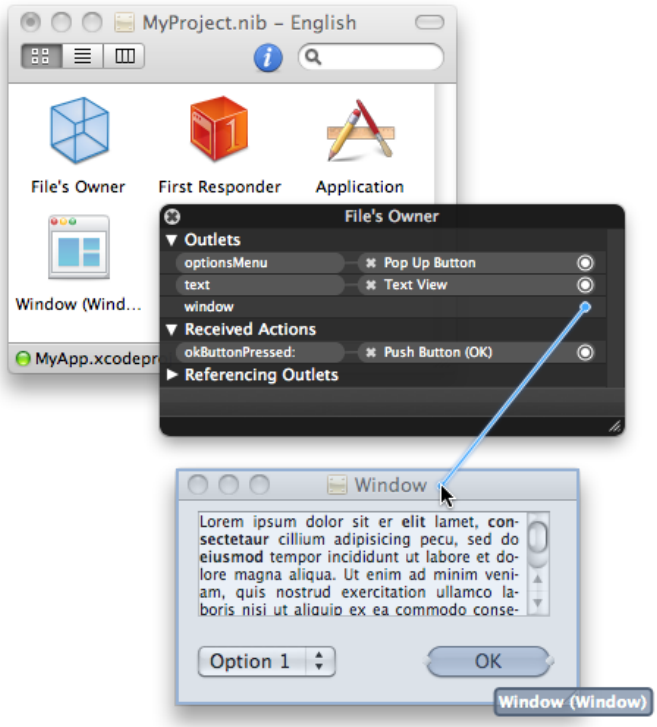


When connecting from target to source, you can connect each action method multiple times. After you make the first connection, the circle next to the action method is filled to show that there is an associated source object. To connect an additional source object, simply click in the circle again and drag to the desired object. Performing this action does not remove the original connection.

Example: Connecting Outlets

Like action connections, you can create outlet connections starting at either the source or target object. Figure 7-3 shows an example of connecting an outlet starting at the outlet's source, which in this case is the File's Owner object. Control-clicking the File's Owner object brings up its connections panel. From there, it is a matter of dragging from the circle next to the window outlet to the window object.

Figure 7-3 Connecting an outlet to the target



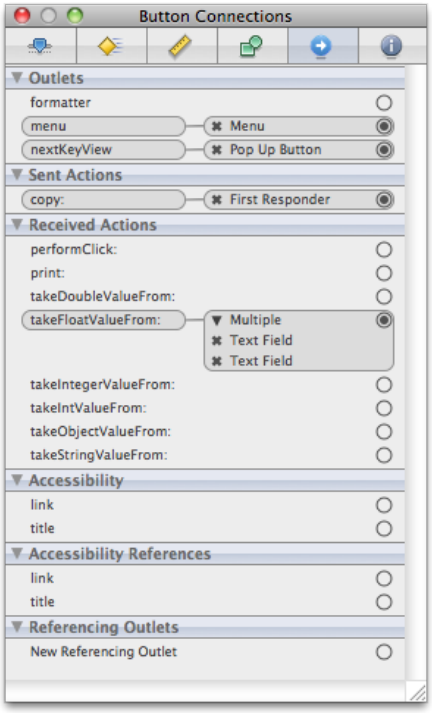
Note: In Mac OS X and in iOS 3.3 and earlier, only one object at a time may be connected to a given outlet. In iOS 3.4 and later, multiple objects can be connected to a given outlet. See "Defining Outlets and Actions in Interface Builder" (page 145) for details.

To connect an outlet starting at the target object, open the connections panel and click in the circle next to the New Referencing Outlet entry. From there, drag to the object that contains the desired outlet and release the mouse over that object. When you release the mouse, Interface Builder prompts you to select the outlet from a list of available outlets on the source object. Selecting an item from that list completes the connection.

Making Connections Using the Inspector

The Connections pane of the inspector window (Figure 7-4) provides a summary of the outlets and actions (or events) of the selected object. You can use the inspector window to view the status of connections, to create new connections, and to break existing connections.

Figure 7-4 Connections inspector



To make a connection, do the following:

1. Select an object and open the connections inspector.
2. Click in the circle next to the entry you want to connect and hold the mouse button down.
3. Drag the mouse over the target object for the connection. (If the target of a connection is not visible, hovering over its parent object causes the parent to open and reveal its children.)

   The target object should highlight to indicate a connection is possible. If it does not highlight, the target object is not of the right type and cannot be connected to the source object in that way.
4. If the target object highlights, release the mouse button to create the connection.
5. If you are using the connections inspector to configure the sent action or referencing outlet for the object, Interface Builder displays a list of the target object's action methods or outlets. Select the desired action method or outlet from the list to finish the connection.

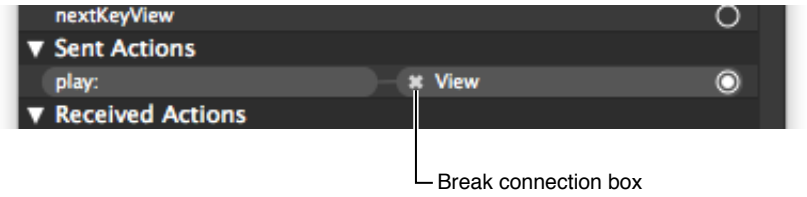
   Other types of connections do not require this configuration step.

After you establish a connection, the connections inspector fills the circle next to that entry and displays information about the connection. Each connection also includes a close box icon that you can use to break the connection. If a received action has multiple source objects associated with it, the panel displays a disclosure triangle, which you can use to reveal the individual connections.

Breaking Connections

To break a connection using the connections panel, do the following:

1. Control-click the desired object to display its connections panel.
2. Locate the outlet or action whose connection you want to break.
3. Click the "Break connection box" next to the name of the connected object.



An alternative to using the connections panel is to break connections using the inspector window. The Connections pane of the inspector window lists all of the connections for the currently selected objects. To break a connection using the inspector window, you locate the connection and click its break connection box. For more information about the connections inspector, see "Making Connections Using the Inspector" (page 124).

Establishing Connections to the First Responder

The First Responder placeholder object acts as a placeholder for an object that is determined dynamically at runtime. The AppKit and UIKit frameworks use several factors to determine which object should be the First Responder, including which window is frontmost, which view has the focus, or which view is designated as the initial responder.

In Cocoa applications, many First Responder actions are initiated by menu items and other objects of your program that operate on the application, the current document, or the frontmost window. Typical actions that are dispatched to the First Responder include the following:

- Document-level operations (such as undo, save, and print)
- Pasteboard operations
- Text-manipulation operations
- Selection management operations (such as select all)
- Application-level operations (open document, hide, unhide, show help)
- Custom actions that you define

In iOS applications, touch events are delivered to the first responder initially and then passed down the responder chain as needed until an object handles the event. In most cases, the first responder is the view in which a touch occurred, but it can be other views or objects in the application.

If you have an action message that should be handled by the First Responder, you can use the First Responder placeholder object as the target object for your action. By default, the First Responder knows about the action messages supported by the system (if any) and those defined in your Xcode source files. If you want to add new action messages, you must add those messages to the First Responder's supported list by doing the following:

1. Select the First Responder placeholder object in your Interface Builder document.
2. Open the inspector window and select the Identity pane.
3. Click the plus (+) button in the First Responder Actions section to create a new action method entry. (The new method name is selected by default.)
4. Type a new name for the method. (The syntax of the method must match the expected syntax for action methods on the given platform; see "Defining Action Methods" (page 143) for more information.)
5. Press Return to save the action method.
6. Repeat as needed to add additional action methods.

Adding action methods to the First Responder placeholder object does not add the corresponding method definition to your Xcode source files. All it does is let Interface Builder know that such a method exists in one of your program's objects. It is up to you to ensure the method names you add to the First Responder placeholder match the names of the methods in your code. Interface Builder does not validate these method names for you. At runtime, if a method name is misspelled or does not exist in an object, the corresponding action message will never be received by the target object.

Note: The First Responder placeholder is only for configuring action messages. You cannot connect the First Responder placeholder object to one of your custom outlets in hopes of receiving a dynamically changing pointer to the selected object in your window. In Cocoa applications, you should instead use the `firstResponder` method of the current `NSWindow` object to get the first responder. In iOS applications, there is no single first responder object; the first responder is always the view that is the target of a touch.

Interface Builder does not prevent you from deleting the standard system messages associated with the First Responder placeholder. Doing so removes the message name from the current nib file only.

Connecting Menu Items to Your Code

There are two common techniques for handling menu commands in a Cocoa application:

- Connect the corresponding menu item to a First Responder method.
- Connect the menu item to a method of your custom application object or your application delegate object.

Of these two techniques, the first is somewhat more common given that many menu commands act on the current document or its contents, which are part of the responder chain. The second technique is used primarily to handle commands that are global to the application, such as displaying preferences or creating a new document. It is possible for a custom application object or its delegate to dispatch events to documents, but doing so is generally more cumbersome and prone to errors.

The behavior for connecting a menu command to an appropriate target is the same as for creating other types of connections. You can use the connections panel or do a quick connection by Control-clicking and dragging from the menu item to the desired target. For information about creating connections using these techniques, see "Making Connections Using the Connections Panel" (page 121) and "Making Quick Connections By Dragging" (page 120).

Note: In addition to implementing action methods to respond to your menu commands, also remember to implement the methods of the `NSMenuValidation` protocol to enable the menu items for those commands.

Configuring Cocoa Bindings in Mac OS X

Introduced in Mac OS X v10.3, Cocoa bindings are a controller-layer mechanism used to synchronize the view and model layers of your Cocoa application. Bindings establish a mediated connection between a view and a piece of data, binding them in such a way that a change in one is reflected in the other. You can use bindings to replace the traditional glue code (action messages and custom synchronization code) that you would normally write to synchronize your application's user interface. For example, you might use bindings to tie a property in one of your custom data objects to a text field in your user interface. As the user changes the value in that field, the bindings code automatically updates the corresponding data object; conversely, if you change the property programmatically, bindings propagate the change back to the text field.

One of the advantages of using bindings over traditional glue code is that you can use Interface Builder to configure them. Objects with bindable properties can expose those properties in Interface Builder through their Interface Builder plug-in object. For each exposed binding, Interface Builder creates an interface that lets the user configure that binding directly using the bindings inspector. The configured bindings are then saved in the nib file and recreated at runtime like other types of connections.

You configure bindings in Interface Builder by starting at the object that exposes a bindable property. Typically, this object is a Cocoa view or controller object, although you can also expose bindable properties from your own custom objects using an Interface Builder plug-in. You then use the Bindings pane of the inspector window to specify the target of the binding and the binding options. You must configure each binding separately and each binding can be attached to a different target object if desired. The target of a binding is always one of the recognized controller objects in your nib file, which typically includes the File's Owner, the application, the shared user defaults controller, and any custom controller objects (especially `NSController` objects) you add to the nib file.

For more information about how Cocoa bindings work, see *Cocoa Bindings Programming Topics*.

Note: Cocoa bindings are not available in iOS.

Creating a Binding

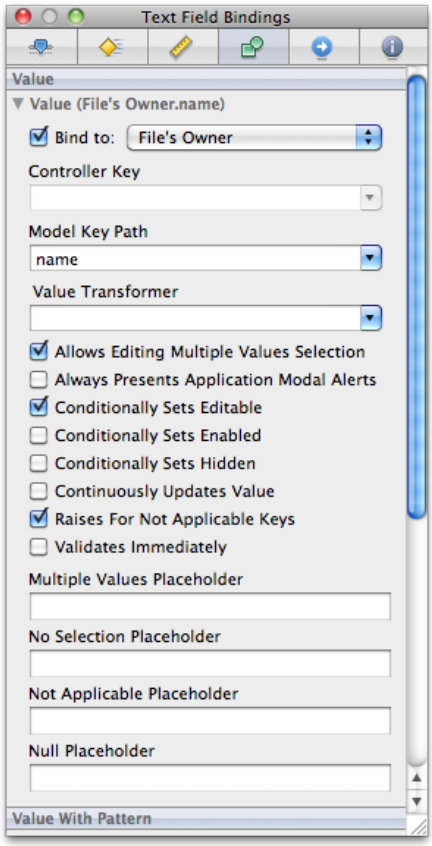
In Interface Builder, you typically bind Cocoa views and controller objects in your nib file to the data in your custom controller objects. The high-level process for creating a binding in Interface Builder is as follows:

1. Create the views needed to display your data.

2. Create any intermediate controller objects needed to manage your data. (Typical controller objects include instances of your custom `NSDocument` or `NSWindowController` subclasses, `NSController` subclasses, or custom `NSObject` subclasses that you create to manage your data structures.)

3. Use the Bindings pane of the inspector window (Figure 7-5) to configure each desired binding.

Figure 7-5 Bindings inspector



Although each binding displays several configuration options, the most important part of a binding is the target of the binding. You must configure, at a minimum, the following fields for any given binding:

- Bind to
- Model Key Path

The Bind to field specifies the controller object to use as the starting point for accessing the target data. The Model Key Path field contains a string representing the key path for the desired data. Key path strings are of the form `<property_name>[.<property_name>]*`. The first property name in this string is a property on the controller object specified by the Bind to field. Each subsequent property name corresponds to a property of the object returned by the previous property name. Say, for example, that you have a custom controller object and the key path string "`person.address.street`". The `person` property returns the person object of the bound controller. The `address` property returns the address of the corresponding person object. And the `street` property returns the desired data value stored in the address object.

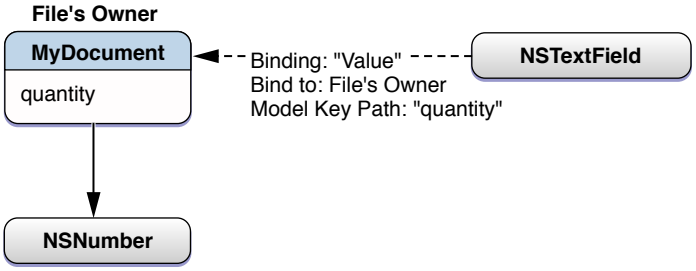
Because key paths are dependent on your data model, the best way to understand how to configure them in Interface Builder is to go through some examples. The following sections show how to bind a text field to an `NSNumber` object so as to display the number value in that text field. In each successive example, the data model used to access the number value gets progressively more complex, and so the bindings must be adjusted to compensate for the increased complexity.

For additional bindings examples, see *Cocoa Bindings Programming Topics*. For information about the key fields in the bindings inspector, see "Configuring the Attributes of a Binding" (page 133).

Binding Directly to the Value

Figure 7-6 shows a simple relationship between a document object and some data. In this case, the document object contains an instance variable called `quantity` that contains a pointer to an `NSNumber` object. (The document also implements the `quantity` and `setQuantity:` accessor methods, which are the KVC-compliant methods for accessing the data.) To display the value from `quantity` in the text field, you would configure the "value" binding for the text field. For this binding, you would set the target object to the document (File's Owner) and the model key path to the string "quantity".

Figure 7-6 A simple bindings example



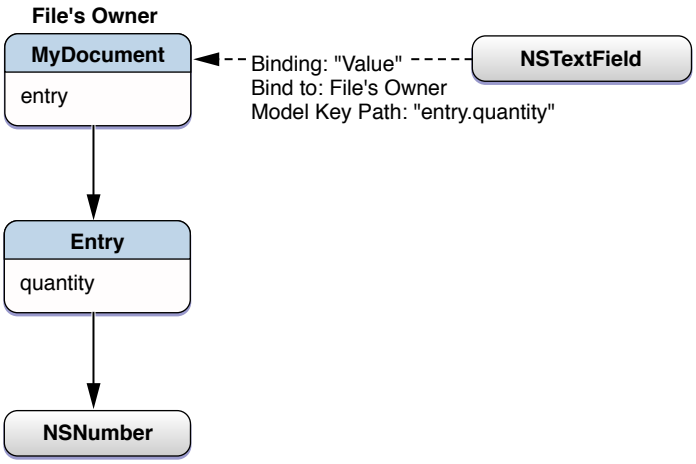
Note: Because Cocoa knows how to convert between many scalar types and their object equivalents, the binding configuration would be the same if you implemented the `quantity` property as a scalar value.

Binding Through an Intermediate Object

The following example builds on the example in "Binding Directly to the Value" (page 130) by introducing an intermediate object between the document and the `NSNumber` object. In this case, the document now contains an `Entry` object, which is a generic data object based on the `NSObject` class.

Figure 7-7 shows the data model relationships for the document and also shows one way to bind a text field to the `quantity` attribute of the `Entry` object. In this case, the text field binds to the document (File's Owner) and has a key path that includes both the `entry` property of the document and the `quantity` property of the `Entry` object.

Figure 7-7 Binding through another data object

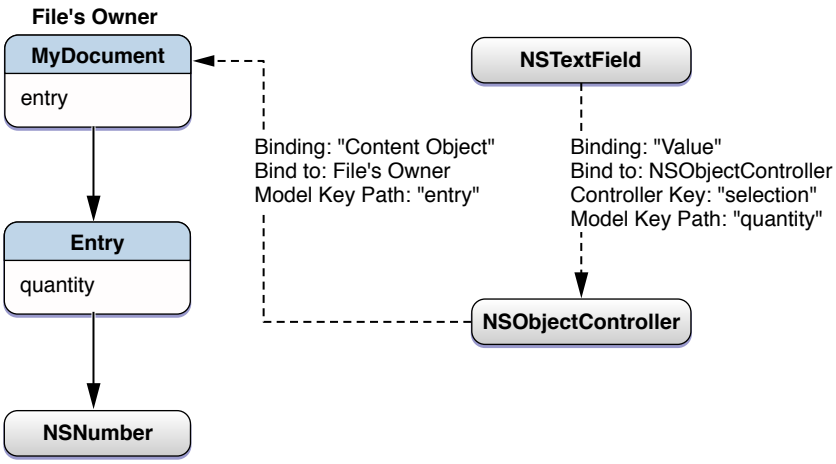


Although the preceding example would work when the document's `entry` property is non-`nil`, a problem arises if `entry` ever becomes `nil`. When Cocoa encounters a `nil` value in the middle of a key path, it generates a runtime error. This error does not abort your program but it does prevent Cocoa from retrieving and setting the bound value, which is certainly undesirable.

Rather than binding the text field directly to the File's Owner, as in the previous example, a better solution is to bind the text field to the value through an `NSObjectController` object. Controller objects act somewhat like a buffer layer between your views and data object. In this case, the controller object acts as a placeholder for the document's current `Entry` object, providing a valid binding target even if the document's `entry` property is `nil`.

Figure 7-8 therefore shows the newly introduced object controller and the revised bindings paths. Now, the text field binds to the object controller as if it were the document's `Entry` object. The object controller, in turn, binds to the `entry` property of the document.

Figure 7-8 Binding through an object controller



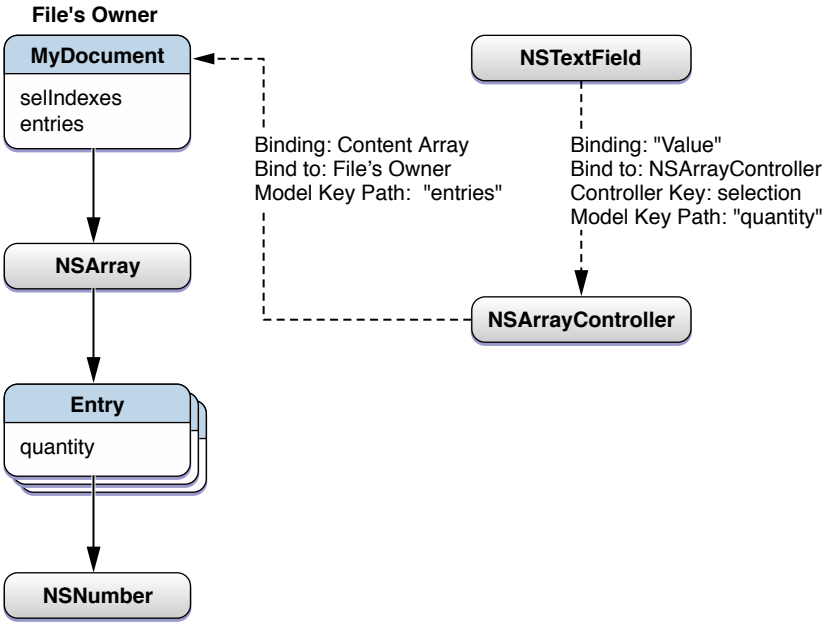
Because it knows about controllers and their properties, Interface Builder displays the bindable properties of the selected controller object in the Controller Key popup menu. You typically set the value of this field to either `selection` or `selectedObjects`, depending on your needs. To finish your binding, you add the target property in the Model Key Path field, which in this case is `quantity`. Cocoa combines the strings in the Controller Key and Model Key Path fields to get the final key path. In this case, the final key path would be `selection.quantity`.

Binding to a Collection of Objects

Using controller objects, it is as simple to bind to a collection of objects as it is to bind to a single object. In fact, the bindings themselves are almost identical. In the example at hand, instead of binding to a single `Entry` object, you would bind the text field to a collection of `Entry` objects through an `NSArrayController` object. The array controller manages both the collection of objects and the list of indexes corresponding to the selected objects and uses that information to provide an appropriate value to the text field.

Figure 7-9 shows the data model for the revised document, which now contains an array of `Entry` objects. You would then bind the text field to an `NSArrayController` object's selection property, which represents the currently selected `Entry` object, and specify the `quantity` string for the key path to finish the binding.

Figure 7-9 Binding to an array of objects



Although the array controller manages the selection indexes for you, you can bind an `NSMutableIndexSet` object to the array controller if you want to keep track of the currently selected indexes. You might store the selected index information as part of your document format. You can also use the index set to change the currently selected objects (instead of using the methods of the `NSArrayController` class). When modifying your bound index set directly, however, you must do so in a KVO-compliant manner by calling the `willChangeValueForKey:` and `didChangeValueForKey:` methods before and after you make your changes. If you forget to call these methods, the array controller may not notice your changes.

Configuring the Attributes of a Binding

Although the bindings inspector may seem complex at first, for many bindings, you need to configure only a few options initially. You may not need to configure every option every time (except for the Bind to and Model Key Path fields). During your initial design phase, you can probably leave most of the options configured with their default values. As you improve your design, you can customize the options more until you have the behavior you want.

Table 7-1 lists some of the key attributes used in most binding entries. Except where noted otherwise, all attributes are optional.

Table 7-1 Common binding attributes

Binding attribute	Description
Bind to:	(Required) The controller object in your nib file that serves as the starting point of the binding. The first entry in the corresponding key path must correspond to a property in this object.
Controller Key	An attribute of an `NSController` object. When binding to an `NSController` object, you use this field to select the first entry in the key path. The menu associated with this field displays the properties available on the selected controller object as a convenience. You can type the name of the property or simply select it from the provided list.
Model Key Path	(Required) The key path to the desired data value. The string you enter into this field is a period-separated list of keywords, each of which represents a property of an object in the path. The objects and properties represented by this key path must exist or Cocoa will be unable to bind to the value. If you specified a value in the Controller Key field, that value is added to the beginning of this string during resolution of the key path. The format of entries in this field is `<property_name>[.<property_name>]*`.
Value Transformer	The name of the value transformer object you want to use to use with this binding. Value transformers let the Cocoa bindings code adjust unknown data formats to a format compatible with the corresponding view. The name you specify in this field corresponds to the name under which your value transformer object is registered in your code, which may or may not correspond to the actual class name. For more information about creating and using value transformers, see *Value Transformer Programming Guide*.
Multiple Values Placeholder	The behavior of a view when the binding is associated with multiple objects. The exact configuration of this field depends on the type of the object exposing the binding. For views and controls, you use this field to specify the default behavior (or value) of the view when multiple objects are selected.
No Selection Placeholder	The behavior of a view when the binding is associated with a valid set of objects but none of them are actually selected. The exact configuration of this field depends on the type of the object exposing the binding. For views and controls, you use this field to specify the default behavior (or value) of the view when there is no selection.

Binding attribute	Description
Not Applicable Placeholder	The behavior of a view when an exception is raised because the key of an object cannot be applied for some reason. This might occur if the specified object is not key-value coding compliant for the specified key.
Null Placeholder	The behavior of a view when the binding is associated with a `nil` object. The exact configuration of this field depends on the type of the object. For views and controls, you can use this field to specify a default value to display.

For an exact list of bindings available for a given view or controller object, see *Cocoa Bindings Reference*.

Using Cocoa Controller Objects

Although you can bind the views of your user interface to any key-value coding (KVC) and key-value observing (KVO) compliant object, it is strongly recommended that you use an intermediate Cocoa controller object to manage the bindings for you. Cocoa controller objects are instances of the `NSController` class that manage bindings-related behavior. These objects provide several very important features, including the following:

- Controller objects manage their own current selection and placeholder values, providing appropriate values if the selection is empty.
- Controller objects implement the `NSEditor` and `NSEditorRegistration` protocols, which provide the associated view with a way to negotiate uncommitted changes between itself and the controller.
- Controller objects eliminate the need for complex management code. Nearly all of your controller object setup is done in Interface Builder.
- Most controllers provide default actions for manipulating the managed content.

Cocoa provides controller objects for managing several different types of data objects. Most of these controllers enable you to manage collections of objects, with the controller object itself managing things like the currently selected objects. Table 7-2 lists the available controller object classes along with information about when you might use each one and how you configure its attributes.

Table 7-2 Cocoa controller objects

Class	Description
`NSObjectController`	Manages any single `NSObject` instance. Use this controller to manage a single custom object and its properties. When configuring the attributes for this controller, the Class Name field (in the Attributes pane of the inspector) should contain the class of the object being managed.
`NSArrayController`	Manages the contents of an `NSArray` or `NSSet` object. (You can also use this controller to manage custom data objects that implement ordered sets of data. For more information, see *Cocoa Bindings Programming Topics*.) An array controller tracks not only the contents of the array or set but also the subset of objects in the set that represent the current selection. When configuring the attributes for this controller, the Class Name field (in the Attributes pane of the inspector) should contain the class of the objects in the array and not the class of the array itself.

Class	Description
`NSTreeController`	Manages a custom set of objects organized in a tree structure. When configuring this controller, you must specify the keys used to access child objects in the tree. In addition, the Class Name field (in the Attributes pane of the inspector) should contain the class of the objects in the tree. For more information, see "Configuring an NSTreeController Object" (page 135).
`NSUserDefaults-Controller`	Provides your application with access to the application's defaults database. You typically use this controller to implement preferences panels by binding the controls in your preferences window to keys in the defaults database. The user defaults controller does not use object class name information. For more information, see "Using the Shared User Defaults Controller" (page 136).
`NSDictionary-Controller`	Introduced in Mac OS X v10.5, this controller manages the keys and values inside an `NSDictionary` object. The controller turns the contents of the dictionary into an array of key-value pairs that can be bound to user interface items, such as columns in a table view. For more information about using this object, see *NSDictionaryController Class Reference*.

When creating bindings in your nib file, you should always use Cocoa controller objects when you are binding through some intermediate object. In other words, unless you are binding directly to a scalar type, you should probably be using a Cocoa controller object to manage the objects in your binding key path.

Note: Cocoa controller objects are not necessary for managing data in Cocoa objects that store scalar data types, including `NSString`, `NSNumber`, and `NSValue`. You can bind directly to those objects as if they were scalar types.

In the Attributes pane of the inspector window, many controllers let you specify a list of key names. These keys represent the properties of the managed object to which clients can bind themselves. Adding keys to this list is a shortcut that eliminates the need to type the key name each time you want to bind to them. Interface Builder displays the key names in the Model Key Path field's popup menu whenever you bind to the corresponding controller object. To finish the binding, simply select the key from this menu.

For more information about object management in controllers, see *Cocoa Bindings Programming Topics*.

Configuring an NSTreeController Object

If your program contains a custom tree data structure, you can use an `NSTreeController` object to coordinate bindings to the objects in that data structure. Tree controllers are often used to bind data objects to an `NSBrowser` or `NSOutlineView` object in order to display hierarchical sets of data to the user.

To configure a tree controller object, do the following:

1. Add the tree controller object to your document and select it.
2. Open the inspector window and select the Attributes pane.
3. Fill in the Object Controller section.
    - In Class Name field, specify the class name of the objects in the tree.

- Optionally, specify any custom keys of the tree objects that you want to expose in the Model Key Path field of the bindings inspector.

4. Fill in the Tree Controller section.
   - In the Children field, specify the key path for the property that identifies the child nodes of a given tree object.
   - In the Count field, specify the key path for the property that indicates how many children a given tree object has. (This key path is optional.)
   - In the Leaf field, specify the key path for the property that identifies whether the current tree object is a leaf node. (This key path is optional but recommended because it can improve the tree controller's performance as it navigates your tree at runtime.)

For more information on using tree controllers, see *Cocoa Bindings Programming Topics*.

Using the Shared User Defaults Controller

An `NSUserDefaultsController` object is most often used when implementing a preferences window or in any situation where you want to bind a view to values in the defaults database. Although you can add a new `NSUserDefaultsController` objects to your nib file, you generally do not need to do so. Interface Builder provides a shared user defaults controller object that you can use to bind to your application's preferences.

To use the shared defaults controller to implement your application's preferences window, you would do the following:

1. Select the control you want to bind to a preference.
2. Open the inspector window and select the Bindings pane.
3. In the appropriate binding for your control, set the Bind to field to Shared User Defaults Controller.
4. In the Model Key Path field, enter the key name of the preference you want to associate with the control.

By default, the shared user defaults controller sets the value in the Controller Key field to "values". Individual preference values are accessed through this property on the user defaults controller, so you should leave this field configured as is.

For more information on configuring user defaults bindings, see "User Defaults and Bindings" in *Cocoa Bindings Programming Topics*.

Copying and Pasting Connections

Copying and pasting objects also copies and pastes outlets, actions, and bindings. If the logical source of the connection is copied and pasted, the paste operation duplicates the connection. Action connections are duplicated to and from the pasteboard with the control but not the controller. Outlet connections are

duplicated to and from the pasteboard with the object that declares the outlet, not the object the outlet points to. Bindings are duplicated to and from the pasteboard with the bound objects, not the controllers the objects are bound to.

Here are some examples:

- Duplicating a table view would duplicate the delegate and data source outlets.
- Duplicating a "My Application Delegate" controller object would duplicate all of its outlet connections.
- Duplicating a table view would duplicate all of its bindings to controllers.

Xcode Integration

Interface Builder itself is not a coding environment; it is a visual design environment. Although you can use it to define new Objective-C classes, and to add outlets and actions to existing classes, the preferred approach is to use Xcode for those tasks. Interface Builder is tightly integrated with Xcode and is able to retrieve information about the classes in a project and make that information available to you as you work on the project's associated nib files.

To make the most of Interface Builder's integration with Xcode, you should do the following during development:

1. Keep your Xcode project open while editing your nib files.
2. Whenever you want to create a new class, or add an outlet or action to an existing class, do it in Xcode.
3. To set the class of an object, simply type its name in the Identity pane of the inspector window.

The Xcode integration works best if both Xcode and Interface builder are running at the same time. If your Xcode project is open, Interface Builder periodically queries it for information about new classes or changes to existing classes. You can tell if a document is associated with an Xcode project by looking at status bar along the bottom edge of the document window. This bar displays the name of the associated Xcode project and an indicator that lights up green when the project is available.

Note: Because it associates each nib file with a specific Xcode project, Interface Builder knows only about the classes that are defined in that project. If the project includes an external framework that defines custom controls or objects, you must add any class definitions from that framework to your nib file manually as described in "Establishing Connections to the First Responder" (page 126).

If Xcode is not running, Interface Builder supports alternate ways to associate your class information with your nib files. The following sections describe these techniques and show you how to use the Xcode integration to build your user interfaces efficiently. The Xcode integration is supported only for applications that use the AppKit or UIKit frameworks and is not supported for Carbon applications. For additional information about working in the Xcode environment, see *Xcode Workspace Guide*.

Setting the Class of an Object

Generic Objects

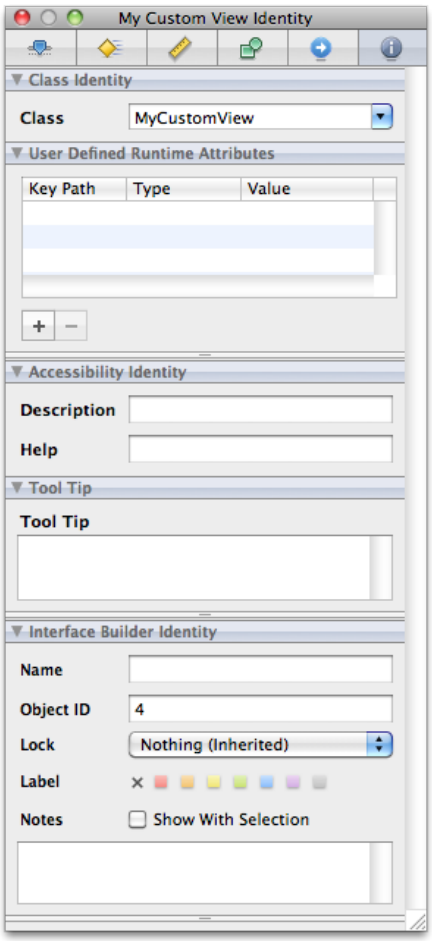
Although most objects in the library are provided by the platform and have a known type, the library also contains generic objects (Figure 8-1) that you can use to represent any class in your project. Using these generic objects lets you incorporate custom code that is not otherwise part of the Interface Builder library. Placeholder objects, like File's Owner, are another type of generic object whose class you must set prior to use.

Figure 8-1 Generic objects

To add a custom view or object to your Interface Builder document, do the following:

1. Drag the desired object from the library.
    - For a generic view, drop it onto your window's design surface. (You can also drop views onto the Interface Builder document window if you just want a view resource without a surrounding window.)
    - For a generic object, drop it into your Interface Builder document window.
2. Select the view or object.
3. Open the identity inspector.
4. Use the available controls to configure the class information.
    - In Cocoa and Cocoa Touch nib files, configure the class name using the Class Identity section of the identity inspector; see Figure 8-2. Type the name of your class in the Class field or use the combo box control to select the class from the current list of known classes.

Figure 8-2 Setting the class of an object



- In Carbon nib files, specify the Class ID of the class in the Class Identity section of the identity inspector.

If your nib file is associated with an Xcode project, Interface Builder attempts to complete the class name automatically as you type it. When the class is set, Interface Builder displays the known outlets and actions for that class in the inspector. If your nib file is not associated with an Xcode project, Interface Builder displays the class name but does not display any outlets or actions initially. You can add outlet and action information using the appropriate sections of the inspector if you like. If you do so, however, you must manually add those same outlets and actions to your source files later. For information about using Interface Builder to define outlets and actions, see "Defining Outlets and Actions in Interface Builder" (page 145).

Custom Objects

Because the Classes library makes it easy to instantiate your custom classes, the recommended workflow is:

1. Create your custom classes in Xcode.
2. Instantiate them in Interface Builder by dragging them out of the Classes library.

This eliminates the need to drag out a generic view or object and then set its custom class.

Defining Outlets and Actions in Xcode

In Cocoa and Cocoa Touch applications, outlets and actions are a way to create dedicated connections among the objects that exist both inside and outside of a nib file. Outlets are essentially instance variables that refer to other objects. Actions are messages that objects send to an associated target object in response to certain events. You identify outlets and actions in your code using the `IBOutlet`, `IBOutletCollection`, and `IBAction` keywords as described in the following sections.

Defining Outlets

The `IBOutlet` and `IBOutletCollection` keywords signal to Interface Builder that you want to populate a variable with real objects when a nib file is loaded. Outlets can be weakly typed or strongly typed in your code. Weakly typed outlets (those whose type is `id`) can be connected to any object in your nib file. Strongly typed outlets can be connected only to an object whose class matches the type of the outlet.

To add an outlet to a class, you can insert the `IBOutlet` keyword in front of its instance variable declaration, as shown in the following example:

```
@interface MyClass : NSObject
{
    IBOutlet id        aGenericOutlet;
    IBOutlet NSView*   aViewOutlet;
}
@end
```

If you use this approach, you should define accessor methods that ensure the objects are retained at runtime.

In Mac OS X and in iOS 3.3 and earlier, you can connect an outlet to only a single object. However, starting with iOS 3.4, you can connect an outlet to multiple objects. To do so, use the `IBOutletCollection` keyword. For example, to connect a view controller to multiple objects of type `UILabel`, you could add the following code in Xcode:

```
@interface MyController : UIViewController {
    IBOutletCollection (UILabel) NSArray* multipleLabels;
}
@end
```

If the output targets are not all the same kind of object, use `id` for the object type in the declaration, or omit the object type in the parentheses following `IBObjectCollection`; for example:

```
@interface MyController : UIViewController {
    IBOutletCollection (id) NSArray* multipleObjects;
}
@end
```

A better approach is to declare each outlet as a property. The `IBOutlet` or `IBOutletCollection` keyword goes immediately before the property's type information and after any parenthetical attributes, as shown in this example:

```
@interface MyClass : NSObject
```

```
{
    id        aGenericOutlet;
    NSView*   aViewOutlet;
    NSArray*  multipleLabels;
    NSArray*  multipleObjects;
}
@property (nonatomic, retain) IBOutlet id aGenericOutlet;
@property (nonatomic, retain) IBOutlet NSView* aViewOutlet;
@property (nonatomic, retain) IBOutletCollection (UILabel) NSArray *mutipleLabels;
@property (nonatomic, retain) IBOutletCollection (id) NSArray *mutipleObjects;
@end
```

For more information about properties, see *The Objective-C Programming Language*.

Defining Action Methods

An action message sent by an object to its target results in the invocation of an action method on the target object. In essence, this is just another way of saying that the object that sends an action message does so simply by calling a method of its target object. Therefore, in order to define an action, you actually define an action method on the corresponding target object. The definition of this method is where you respond to the action. For example, if the user clicks or taps a button, the corresponding action method is where you would write the code to respond to the button interaction.

In Cocoa applications, an action method takes a single parameter of type `id` and returns the type `IBAction`, which is defined as `void`. The name of the action method can be anything you want but is typically evocative of the action being performed. For example, to respond to a click in a button, you might define the following method in the controller object that manages the button:

```
- (IBAction)respondToButtonClick:(id)sender;
```

In Cocoa Touch applications, an action method can take one of three forms, shown here:

```
- (IBAction)respondToButtonClick;
- (IBAction)respondToButtonClick:(id)sender;
- (IBAction)respondToButtonClick:(id)sender forEvent:(UIEvent*)event;
```

Action methods do not require a one-to-one correspondence with the controls in your interface. You can define one action method for each control in your interface or several controls can share a single action method. When present, the *`sender`* parameter in an action method contains the object that sent the action and can be used to help process the action. In iOS, the optional *`event`* parameter conveys additional details about the specific type of event that triggered the action.

If you subclass standard system classes, you should check the definitions for any parent class before adding your own custom action methods. For example, many Cocoa classes already implement `cut:`, `copy:`, and `paste:` action methods to respond to pasteboard actions. Using inherited action methods lets you respond to messages sent by your own code and also messages sent by the system. In general, you should implement custom action methods only when the action methods provided by parent classes are insufficient or do not provide the behavior you need.

Synchronizing With Your Xcode Project

If your Interface Builder document is already associated with an Xcode project, you do not have to do anything to synchronize the two. When you change your source files in Xcode and save those changes to disk, Interface Builder automatically detects those changes and updates its internal copy of the information. In other words, if you add an outlet to a custom view in Xcode and then switch to Interface Builder, that outlet will appear automatically in the connections panel and inspector windows for that view. Of course, the Xcode syncing support is contingent upon the ability to parse your code successfully. If your code contains syntax errors, Interface Builder may not be able to read the changes.

If you disable the automatic syncing option in the Interface Builder preferences, you can sync your Xcode project manually by choosing File > Reload All Class Files. This command forces Interface Builder to reread any source files containing relevant class definitions.

Creating Classes in Interface Builder

Although Xcode is the preferred environment for creating new classes, there may be cases where you want to define your nib file first, including the definitions for any custom classes. The following sections show you how to do this in Interface Builder.

Note: The following sections are relevant for Cocoa and Cocoa Touch applications only.

Defining New Classes in Interface Builder

The ability to reference unknown classes in Interface Builder lets you rapidly prototype your user interface without worrying about the underlying code. You can change the class name and change the names of outlets and actions as needed. Once you are satisfied with your prototype, you can generate source files for any unknown classes from Interface Builder and begin writing the code for those classes. After generating the source files, you can also specify the superclass for your class. For information about generating class files, see "Generating Source Files for New Classes" (page 147).

Interface Builder 3.2

In Interface Builder 3.2 and later, to define a new Objective-C class do the following:

1. In the Library window, click the classes tab and select the class you want to subclass.
2. From the action menu, choose New Subclass.
3. Enter the name of the subclass and whether or not to immediately generate source code files.

   If you plan to add outlets and actions to the subclass, wait and generate the source code files later.
4. Click OK (or press Return) to define the new class.

Interface Builder 3.1 and Earlier

In Interface Builder versions earlier than 3.2, to define a new Objective-C class do the following:

1. Select the object whose class you want to set.
2. Open the inspector window and select the Identity pane.
3. In the Class field, type the full name of the class.
4. Press Return (or select another field) to apply the class name.

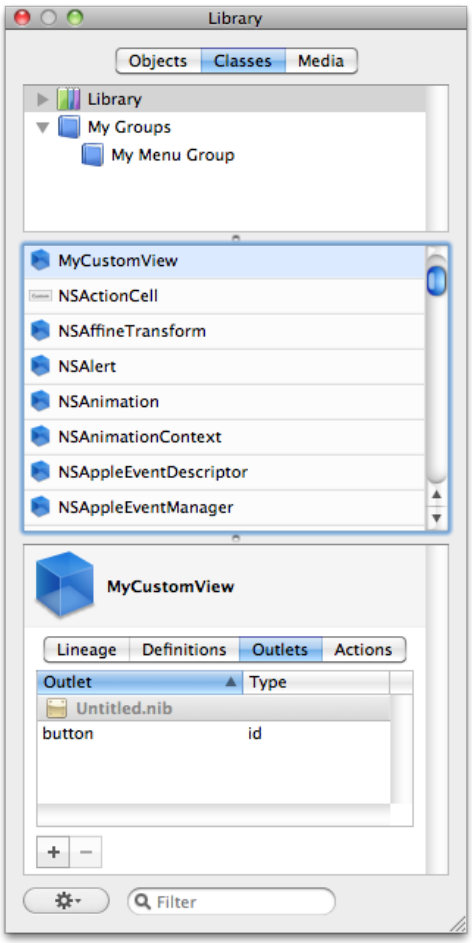
When you type the name of an unknown class in the identity inspector, Interface Builder assumes that this class exists somewhere outside the nib file. After creating the class, you can define outlets and actions for it as described in "Defining Outlets and Actions in Interface Builder" (page 145) and use those outlets and actions to create connections in your nib file. Classes created in this manner have an unknown superclass and exist only within Interface Builder.

Defining Outlets and Actions in Interface Builder

When present, the Class Outlets and Class Actions sections of the Library window display the known outlets and actions of the currently selected class. In addition to viewing the existing list of outlets and actions, you can also add new ones using the controls found in these sections. You would typically use this feature during prototyping, when your nib file does not yet have an associated Xcode project. For example, you might create a new class for your prototype and add outlets and actions to get a sense of what behaviors you would need in your code.

Note: Beginning in Interface Builder 3.2, the management of actions and outlets and been moved from the identity inspector to the classes tab in the Library window. The only exception to this is the First Responder. Actions and Outlets for the First Responder have been moved to the attributes inspector.

Figure 8-3 Defining outlets and actions in the Library window



To add an outlet or action, do the following:

1. Click the plus (+) button in the appropriate section to add the new action or outlet.
2. Type a new name.

   As you type the name of an action method, Interface Builder displays warnings in the Class Actions section of the inspector to let you know if you are specifying an invalid method name.

   In Mac OS X, action methods take a single parameter and therefore the method name must end with a colon (":") character.

   In iOS, action methods may take zero, one, or two parameters.
3. Optionally, specify a custom type for outlets by typing the appropriate class name. (You can also change the parameter type of action methods, although `id` is the standard parameter type.)

To remove a custom defined outlet or action, select it in the identity inspector and click the minus (-) button. You can remove only those outlets and actions you previously added using Interface Builder. You cannot remove outlets and actions you created in your Xcode source files or those defined in a Cocoa parent class.

The outlets and actions you add using the Library window remain local to Interface Builder and your Interface Builder document. Interface Builder does not attempt to merge these outlets and actions back into your Xcode source files; you must do that manually. You can write out source and header files containing the outlet and action definitions for your class by choosing File > Write Class Files. You can then use the generated files as your initial source or merge their contents in with your existing source files using your favorite merge tool, such as the FileMerge application.

Note: To copy outlets and actions into your Xcode source files, you can drag entries from the Library window and drop them into the corresponding source file. Xcode pastes a properly formatted outlet or action declaration into the source file at the drop point. You can also select an outlet or action entry in the Library window and select Edit > Copy to copy the declaration to the pasteboard.

Generating Source Files for New Classes

If you create new classes using the Library window, you can ask Interface Builder to create an initial set of source files for those classes. When generating source files, Interface Builder also generates the appropriate set of member variables and methods corresponding to the outlets and actions of the class. This feature helps your prototyping efforts by eliminating the need to recreate your class definitions manually in Xcode.

To generate source files for a class you defined in Interface Builder:

1. Select the object that uses the desired class.
2. Choose File > Write Class Files.

   Interface Builder prompts you to save the new source files.
3. Open the header file.
4. Specify the superclass information for the class.

Beginning in Interface Builder 3.2, you can generate or update source files using the classes tab in the Library window:

1. Select the class for which you want to generate or update source files.
2. Display the action menu and choose Write Updated Class Files or Generate Class Files.

After generating the source files, you should add them to your Xcode project and make any future changes from there. Changes made to source files in your Xcode project are automatically picked up by Interface Builder, but the reverse is not true. Interface Builder does not update source files after it generates them. Instead, you must incorporate any additional changes into the source files manually from Xcode.

Injecting Class Information into a Nib File

As much as possible, Interface Builder relies on Xcode to provide it with information about your project's custom classes. If you are writing a Cocoa or Cocoa Touch application and that information is not available, however, you can provide it manually by doing one of the following:

- Drag the class header files from Xcode or the Finder and drop them onto your Interface Builder document window.
- From Interface Builder, choose File > Read Class Files and select the header files.

When you use either of these techniques, Interface Builder parses the provided header files for information and caches that information in your Interface Builder document. The cached information remains valid until the next time you import header files or until Interface Builder accesses the class information from the associated Xcode project.

If the header files you import refer to other custom classes, you should import the header files for those other classes as well. This is especially important if the parent of your custom class is also a custom class. Without the full parent class hierarchy, Interface Builder cannot build a complete picture of the outlets and actions of the class. However, you do not need to import classes defined in the AppKit or UIKit frameworks or those that are defined in an Interface Builder plug-in.

Scripting Language Support

You can use Interface Builder for more than just native Mac OS X and iOS applications. Interface Builder also provides support for applications built using the Ruby, Python, and AppleScript scripting languages and can parse script files that contain the appropriate keywords.

If you are a Ruby, Python, or AppleScript developer, you can take advantage of the Cocoa scripting bridge to write your application code and use nib files to create your application's user interface. As you build your interface, you associate elements from that interface with objects in your scripts much like you would associate them with Objective-C objects. For information about how to use the Cocoa scripting bridge, including the keywords you need to add to your script, see *Scripting Bridge Programming Guide* and *Ruby and Python Programming Topics for Mac OS X*.

Testing and Validation

Interface Builder provides tools to help you validate the contents of your documents against the platform on which you plan to use them. In addition to testing the behavior of your interface in the simulator, you can also verify that the operating system on which you plan to run your software supports all of your nib file objects. Different versions of an operating system may support different objects or different attributes for those objects. If your nib files contain objects that are not available on the current platform, your program may crash.

The following sections show you how to use Interface Builder to test your nib files.

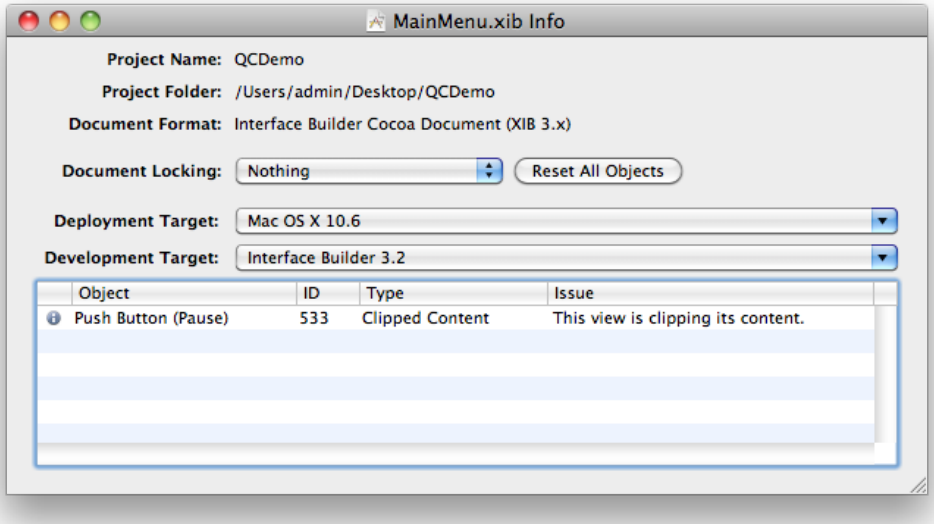
Setting the Nib File Version

To display general information about a nib file, open the nib information window by choosing Window > Document Info. Among other things, this window displays the following pieces of information:

- The associated Xcode project information
- The document format (nib or xib)
- The lock attributes for the nib file; see "Locking Down Your Nib File" (page 153)
- The intended deployment and development targets
- Any design errors, warnings, and notes

Figure 9-1 shows an information window with a design time issue. Double-clicking an item in this table selects the item and brings it to the foreground.

Figure 9-1 Nib information window



For each nib file in your project, you should:

- Set the deployment target value to match the deployment target of the Xcode target that uses the nib file.

  Interface Builder uses deployment targets to identify potential problems with your nib file. For example, if your deployment target is set to Mac OS X v10.3 but a window includes a control that was introduced in Mac OS X v10.5, Interface Builder reports the inclusion of the control as an error. Correcting deployment errors ensures that the nib file can be loaded successfully on the target platform.

  The deployment target has a default setting which you can "set and forget." The default setting is tied to the associated Xcode project's SDK setting. When you change the project's base SDK, the deployment target of the NIB file changes automatically.

- Set the development target value to indicate which versions of Interface Builder can use the file.

  If the development target is set to 3.2, the file will be usable with Interface Builder 3.2 and later. If the target is set to 3.0, the file will be usable with Interface Builder 3.0, 3.1, and 3.2. For example, if your development target is set to 3.0 and you have a custom object with user-defined runtime attributes (a feature introduced in 3.2), Interface Builder reports an error. Correcting development errors ensures that the nib file can be edited successfully with the target version of Interface Builder.

You must remove errors before saving your nib file and should try to remove warnings as well. Although you can change the severity of different types of design issues from the Interface Builder preferences window, you should do so only when you know that it will not affect the ability to load and use your nib file.

Checking for Errors and Warnings

Prior to saving your nib file, it is a good idea to set the deployment target and see whether the file generates any errors or warnings. Although Interface Builder does not prevent you from saving a nib that contains errors and warnings, those conditions may cause problems when you load the nib into your application. For example, if your nib contains controls that were introduced in Mac OS X v10.4, your nib-loading code would not be able to load the nib on a computer running Mac OS X v10.3.

To view the errors and warnings in your nib file, you use the information window (see Figure 9-1 (page 150)). From this window, you can set the intended deployment target for the nib file and see whether it contains an errors or warnings that might prevent the nib file from loading. Double-clicking the entries in the table takes you directly to the object that is causing the problem.

Simulating Your Interface

Although Interface Builder displays your views almost exactly as they appear in your application, it is primarily intended for designing interfaces and therefore adds adornments and other visual cues to help you manipulate your views and controls. The simulator presents your views exactly as they would appear when loaded by your application. In addition, the simulator provides the following behaviors:

- The resizing rules of your window work exactly as they would at runtime.
- Cocoa bindings are active. Any views bound to user defaults or other views display the appropriate data.
- Actions that occur relative to other views in the window can be exercised.

To enter simulation mode for a window, select the window and choose File > Simulate Interface. To exit simulation mode, choose Quit from the simulator application menu or press Command-Q. Upon quitting the simulator, you are returned to Interface Builder.

If your interface is designed for an iOS device, you may want to simulate the interface for a different device, iPad or iPhone. For an iPad interface, choose File > Simulate as iPhone/iPod touch. For an iPhone interface, choose File > Simulate as iPad.

Localization

After you finish your application's user interface, you can hand off completed nib files to your localization team for translation into other languages. The localization team can use the `ibtool` command-line program to extract any localizable strings from your nib files and create matching nib files in your product's target languages. The team can also use Interface Builder to lock nib files to prevent engineers from making further changes that might impact in-progress localizations. The following sections discuss how to use these features in Interface Builder.

Locking Down Your Nib File

When you finalize the contents of a nib file in your development language, you can tell Interface Builder to lock the contents of that nib file. Locking a nib file prevents users from making accidental changes that might impact localization efforts. When locked, the inspector window displays all relevant properties of the object as read-only. For properties that can be changed by direct manipulation, Interface Builder also prevents the user from editing those properties directly.

Interface Builder supports several different levels of locking to give you flexibility on what can and cannot be edited in your nib file. Table 10-1 lists the available locking options and what impact they have on your objects.

Table 10-1 Locking options for nib-file objects

Locking option	Description
Nothing	All properties of the object are modifiable.
All properties	All properties of the object are locked.
Localizable properties	The user may not change any user-visible strings and may not change a limited set of other attributes, such as the object's size. The user may make other changes, though. For example, the user could change the enabled state of a control or cell.
Non-localizable properties	The user may change user-visible strings and attributes such as the size of the object, but may not change any other attributes of the object. The localization team can use this mode to incorporate translations without accidentally changing other object attributes.

You can apply lock attributes to your entire nib file or on an object-by-object basis. By default, nib-file objects are configured to inherit their lock attribute from their parent object, with top-level objects inheriting their lock attribute from the nib file itself. Changing the lock attribute for a given object affects the selected object and all of its children. Thus, you can use this behavior to lock specific portions of your user interface while still allowing you to work on other portions.

To change the lock attribute for a single object, do the following:

1. Select the object.
2. Open the inspector window and select the Identity pane.
3. In the Interface Builder Identity section, select the desired locking option from the Lock pop-up menu.

To change the lock attributes for your entire nib file, do the following:

1. Select the nib document window.
2. Select Window > Document Info to open the information panel.
3. Select the desired locking option from the Document Locking pop-up menu.

If you want to remove any custom lock attributes from the objects in your nib file, you can do so manually by modifying the individual objects, or you can use the Reset All Objects button in the nib information window. Clicking this button returns all nib-file objects to their default state, whereby they inherit their lock attribute from their parent object. You can use this button to ensure that all nib-file objects are set to a known state.

Localizing Your Nib File's Content

Because the contents of nib files are seen by end users, you must localize them along with your other project resources. The localization process for nib files involves the following steps:

1. From the original nib file, extract the set of localizable strings in that nib file.
2. Translate a copy of the strings file into the target language.
3. Merge the translated strings file back into a new language-specific version of the nib file.
4. Go back through each language-specific nib file and make any needed layout adjustments to account for changes in string length.

The following sections describe the steps for extracting the localizable strings from your nib file and for reincorporating them later. This process is accomplished using `ibtool`, which is a command-line program used to manipulate nib files. This tool is included with Xcode 3.0 and later, and is located in the `<Xcode>/usr/bin` directory.

Note: The `ibtool` program is a reworked version of the `nibtool` command-line program, which is included with earlier versions of Xcode. The `nibtool` program can only parse nib files created by Interface Builder 2.5.x and earlier. The `ibtool` program is capable of parsing Interface Builder 3.0 documents as well as earlier nib file formats. If you're using Interface Builder 3.0 or later, you should convert your 2.x format nib files to one of the 3.x formats before you localize them.

Each time you run it, `ibtool` outputs a status report to the Terminal window. The status report contains any information you requested about the nib file along with any errors that were encountered. The output is formatted as an XML-based property list file, which you can copy and paste into a text file with a `.plist` file and then open using the Property List Editor application (located in `<Xcode>/Applications/Utilities`). For more about the types of information you can include in this property list, see the `ibtool` man page.

The `ibtool` program supports the XLIFF format when working with strings files. XLIFF is a file format than is used to exchange information between many localization tools. For more information about this format, see XLIFF 1.2 specification.

Performing the Initial Localization of a Nib File

To extract the set of localizable strings from your nib file, you use a command similar to the following in Terminal:

```
ibtool --generate-stringsfile MyNib.strings MyNib.nib
```

The `--generate-stringsfile` option causes `ibtool` to parse the specified nib file and generate a strings file containing all of the strings that should be localized, including things such as control titles, tool tips, placeholder text, and accessibility information. The strings file is written out using the UTF-16 encoding, which is the recommended encoding for strings files. For each entry, the file includes comments indicating the object that contains the string, the object's ID, and the current string if one is set.

After you translate the exported strings, you use `ibtool` to merge the translations back into your nib file. In Terminal, you should invoke `ibtool` using a command similar to the following:

```
ibtool --strings-file MyNib.strings --write MyNewNib.nib MyNib.nib
```

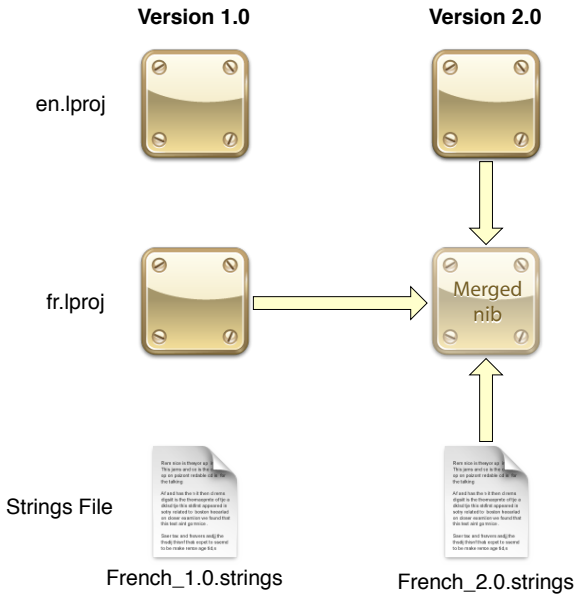
Using these parameters, `ibtool` copies the specified nib file, merges in the contents of the strings file, and writes out a new nib with the changes. When using the `--strings-file` option, `ibtool` updates only those strings that are listed in the specified strings file. If you added any strings to your nib file after extracting the initial strings file, those new strings are not touched during the merge. The `--write` option specifies the name of the new nib file to write out with the localized strings.

Performing Incremental Localization Updates

Localization is a time-consuming and potentially expensive process. A way to save both time and money during localization is to reuse your previously localized content as much as possible. If you make changes to one of your program's master nib files, you can use `ibtool` to merge your changes into the localized versions of the same nib file. This technique is particularly effective for incorporating minor changes to your layout because it saves having to modify each of your translated nib files by hand. If the updated nib file includes new content (including new strings or new localizations of old strings), you can also merge any new translations into the nib files at the same time you make your other layout changes.

Figure 10-1 shows the files involved in an incremental localization update. In this example, changes were made to the English nib file in version 2.0 of the program. The updated English nib file also contains a handful of new controls, the strings for which have been translated into French and incorporated into an updated version of the French strings file.

Figure 10-1 Merging changes into a localized nib file



During the merge, `ibtool` pulls information from both the original and the new versions of the English nib file, plus the original French nib file and updated strings file, to create the new French nib file. The command you would use in Terminal to perform this merge would look similar to the following (without the extra formatting):

```
ibtool --previous-file ./old/Resources/en.lproj/MyNib.nib
       --incremental-file ./old/Resources/fr.lproj/MyNib.nib
       --strings-file ./new/Resources/fr.lproj/French_2.0.strings
       --localize-incremental
       --write ./new/Resources/fr.lproj/MyNib.nib
       ./new/Resources/en.lproj/MyNib.nib
```

The `--previous-file` and `--incremental-file` options specify the master and translated nib files from the previous release. In this case, these are the original English nib file and its French counterpart. The input file for `ibtool` is the updated master nib file containing the new content for the new version of your application. The `--strings-file` parameter is optional and only necessary when you want to incorporate updated or new translations into the merged nib file. The `--write` option specifies the name of the file you want to create based on these inputs.

As it performs the merge, `ibtool` starts with a copy of the new master nib file and then migrates changes from the older translated nib file based on a fixed set of rules. Using the English and French nib files from Figure 10-1 (page 156) to provide context, these rules are as follows:

1. The `ibtool` program copies version 2.0 of the English nib file and uses that as the starting point for the new version 2.0 French nib file.

2. The `ibtool` program compares the objects in versions 1.0 and 2.0 of the English nib files.

- If the properties of an object are the same in both versions of the English nib file, `ibtool` copies the corresponding object from version 1.0 of the French nib file into version 2.0 of the French nib file (replacing the English version of that object). The assumption here is that if nothing changed between the English versions, the object from the older French nib file can be used as is.
- If version 2.0 contains an object that is not present in version 1.0, `ibtool` does nothing. (This means that the version 2.0 of the French nib file contains the new object but with English content.)
- If version 1.0 contains an object that is not present in version 2.0, `ibtool` does nothing. (The object was removed, so it does not need to be merged into the new nib file.)

3. If a strings file was specified, `ibtool` merges the strings into version 2.0 of the French nib file.

When the merge is complete, what you receive is a new nib file that contains all of your new views and controls but also contains as much translated content as possible. Any new objects you added to your interface must still be translated, assuming you did not provide translations in a strings file during the merge, but at least the number of objects requiring translation is reduced.

Warning: Because `ibtool` operates only on the objects found in your master nib files, you should limit the changes you make to localized nib files to changes in strings and geometry only. You should never add objects to, or remove objects from, your localized nib files. You should also not change the type of an object. Such changes are lost when performing an incremental update.

Interface Builder Customization

Interface Builder supports a number of customizations to its workflow environment that are designed to make working with the tool easier. Most of these changes are designed to help you find the information you need quickly. This chapter covers some of the more prevalent techniques designed to make working with the library easier. It also covers the basic Interface Builder preferences and explains how you can integrate your custom objects into the Library window.

Customizing the Library Window

The Library window can be customized in numerous ways to make it easier to find the objects you need. These customizations can be temporary (as in the case of filtering) or persistent (as in the case of custom folders). The following sections describe these customizations and how you make them.

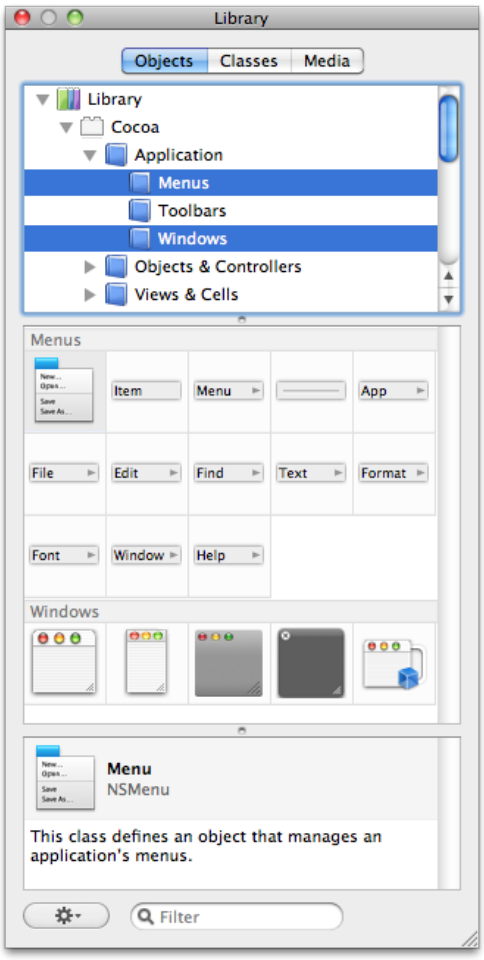
Filtering the Grid Display

There are two ways to limit the number of items displayed in the Library window:

- Select a subset of library groups.
- Use the filter control at the bottom of the Library window.

In the groups pane of the Library window, the Library group contains all of the objects currently integrated into Interface Builder. The contents of the Library group are further subdivided based on the currently loaded plug-ins. Each plug-in can provide further groupings for the controls it defines. Selecting one or more of these subgroups limits the items displayed in the items pane to those in the selected groups. Figure 11-1 shows the window with two groups selected and the resulting list of items that are displayed.

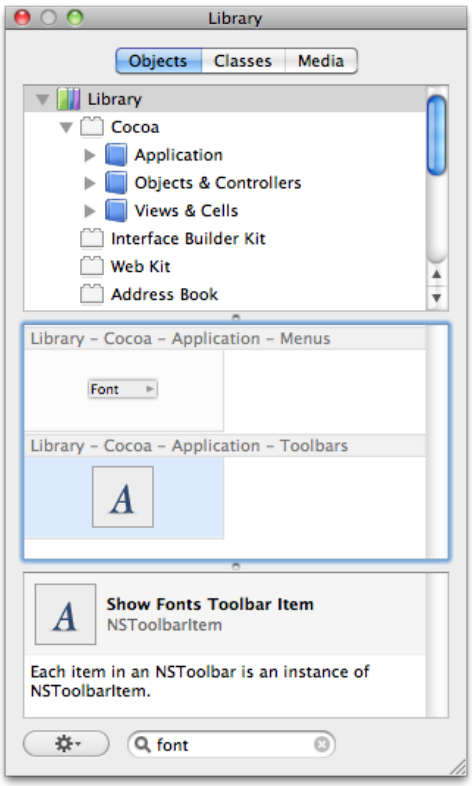
Figure 11-1 Selecting multiple groups



To select a group, simply click it. To add groups to the current selection, hold down the Command key and click on each group. To select a range of groups, hold down the Shift key when clicking.

You can also filter the list of displayed objects using the filter control at the bottom of the Library window. The filter control always operates on the entire library and cannot be combined with other filtering options. Typing the name of an object or a textual description of it restricts the current list of items to those that match the specified string. For example, typing the word `Font` in the default library displays two items, as shown in Figure 11-2.

Figure 11-2 Filtering the contents of the Library window

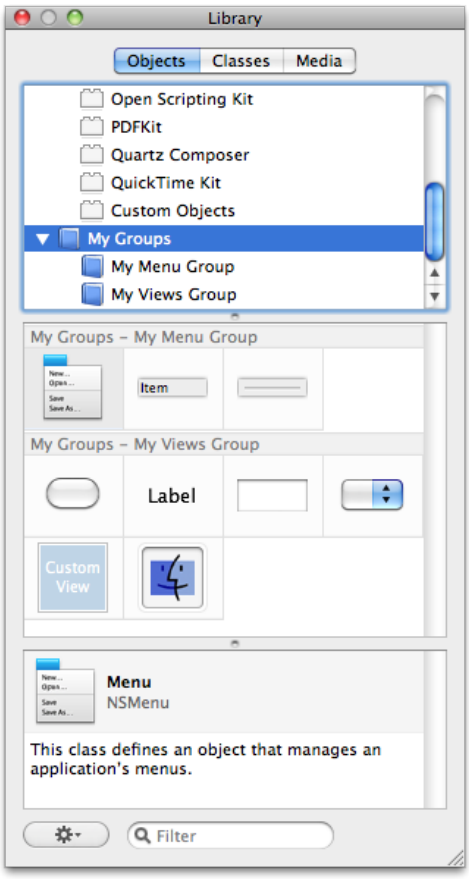


Creating Custom Groups

In addition to the predefined groups provided by each plug-in object, you can create custom groups to organize objects any way you want. To create a new custom group, choose New Group from the action menu at the bottom of the Library window. To add objects to your group, select the Library group (or any of its subgroups) and drag items from the items pane into your group. You can mix objects of any type inside a custom group.

You might use custom groups to store a set of frequently used objects or to implement a different organization for the contents of the main library. Custom groups always appear below the main library, as shown in Figure 11-3. You can arrange your custom groups however you want (including hierarchically) by dragging them around. As with the Library group, selecting a custom group displays the objects in that group plus the objects in its contained child groups.

Figure 11-3 Custom groups in the Library window



Note: Although you can drag objects between custom groups, doing so moves the object from one group to the other. To keep the item in both groups, hold the Option key while dragging or drag the item from the Library group.

Creating Smart Groups

Smart groups are custom groups whose contents are determined by a set of dynamic filtering criteria. You can use smart groups to track recently used objects or objects from across the library that match specific criteria. Because the filtering criteria is dynamic, Interface Builder regularly updates the contents of your smart groups to reflect the current environment.

To create a new smart group, choose New Smart Group from the action menu at the bottom of the Library window. You must specify the filtering criteria for your smart group when you create it. Figure 11-4) shows the rule editor sheet that you use to specify your filtering criteria. As you edit your rules, the sheet displays the current number of objects that match your criteria.

Figure 11-4 Creating smart groups

You can build smart groups to accommodate a variety of conditions. For example, you can build groups that filter based on the object's description or type, or you can build a smart group that displays the objects you used most recently. Table 11-1 lists the types of rules you can configure in the rule editor and describes how Interface Builder matches objects using those rules.

Table 11-1 Rule editor search criteria

Search type	Description	Example
Label	Matches the specified string against the label of the object. You can specify whether the label should start with, end with, or contain the specified string.	Searching the string "Font" yields the "Font Menu item" and the "Show Fonts Toolbar item".
Used Within	Matches objects that were dragged from the Library window within the specified time period. You can use this criteria to show frequently used items.	Specifying a setting of 1 day yields the objects dragged from the library within the most recent 24-hour period of time.

Search type	Description	Example
Search Criteria Matches	Matches the specified string against the label or class name of the object. The object must simply contain the string to match.	Specifying the string “NSP” matches objects with types such as `NSPopUpButton`, `NSProgressIndicator`, and `NSPredicateEditor`.
Categories	Matches the specified string against generic keywords indicating the type of an object. You might use this criteria in combination with others to limit filters to a particular type of object.	Specifying the string “toolbar” with the “Contains Something Like” operator matches toolbar and toolbar item objects.
Is a Kind Of	Matches objects against the specified class name.	Specify "NSControl” to match only those objects that are descendants of the `NSControl` class. You must specify whole class names and not just part of a class name.

Removing Groups and Smart Groups

To remove a group or smart group from the Library window, select the group in the groups pane of the Library window and do one of the following:

- Press the Delete key.
- Choose Remove Group from the action menu.

Rearranging Objects in the Items Pane

You can rearrange objects in a custom group by dragging them around in the items pane. As you drag an item, the items pane provides feedback as to the proposed new location of the item. When dropped, the pane animates the position change of the affected objects.

You can move objects only within custom groups that contain no child groups of their own. You cannot rearrange objects in smart groups or in any of the main library groups regardless of hierarchy.

Adding Custom Objects to the Library

Although you normally drag items out of the Library window and into your user interface, you can also drag your custom configurations of those objects back into the Library window. Doing this lets you retrieve those custom objects later without having to reconfigure them.

To add a custom object to the library, do the following:

1. In your window, configure the object as you would like it.
2. Press and hold the Option key and drag the object to the Library window.

3. In the organization pane of the Library window, drop the object on the Custom Objects group or on a custom group you created. Interface Builder prompts you for information about the dropped object.

4. Fill in the information about your object and press OK.

You can use this technique to drag one object or a group of objects. When dragging more than one object, the entire group becomes a single item in the Library window. Dragging that item back out of the library creates all of the original objects.

The items you add to the library persist between Interface Builder sessions so that you can use them over and over again. To remove a custom item from the Library window, do the following:

1. Select the Custom Objects group in the library to see the items in that group.

2. Select your custom item.

3. Press the Delete key.

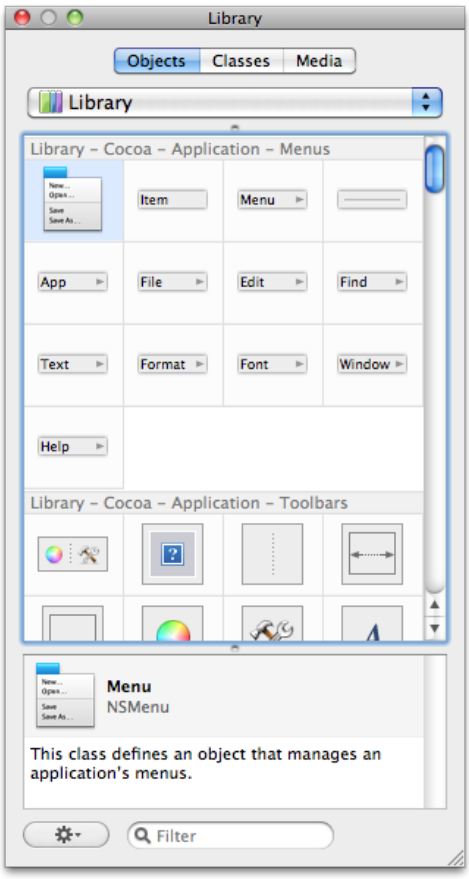
You must remove custom items from the Custom Objects group in order to remove them from the Library window. Removing items from your custom group folders removes them from the group but not the library.

In addition to adding custom configurations of objects to the library, you can also add entirely new objects to the library through an Interface Builder plug-in. Plug-ins are typically used in situations where you want to be able to configure and edit the attributes of your custom classes. For more information, see "Using Plug-ins to Integrate New Objects into the Library" (page 167).

Minimizing the Organization Pane

You can make room for more items and descriptive information in the Library window by minimizing the organization pane. To do this, drag the split bar for that pane up to the pane's top edge. As you near the top, the pane is automatically replaced by a pop-up menu; see Figure 11-5. You can continue to select the currently displayed group using the pop-up menu. Doing so, however, limits you to selecting one group at a time.

Figure 11-5 Minimizing the organization pane



To return to the full organization pane, drag the split bar back down until the pane appears.

Interface Builder Preferences

The Interface Builder preferences window displays application wide settings and configuration options. You use this window to customize the basic application behavior and to load plug-ins containing custom controls. Table 11-2 lists the configuration options found in the preferences window and their purpose.

Table 11-2 Interface Builder preferences

Preference pane	Description
General	Contains preferences for document handling.
Plug-ins	Displays the list of plug-ins currently loaded into the Interface Builder application. You can also use this pane to install or remove plug-ins containing custom controls.
Alerts	Displays preferences for deciding what constitutes errors, warnings, and noteworthy events in a nib file.

Preference pane	Description
Simulator	Displays preferences for managing the simulator user defaults cache. Objects can use the simulator user defaults as a source of initial data. For example, the user defaults controller object takes information from the user defaults database.

Note: If your Xcode project includes a framework that contains an Interface Builder plug-in, Interface Builder loads the plug-in automatically when you open a nib file from that project. You do not need to load it manually using the preferences window.

Using Plug-ins to Integrate New Objects into the Library

Interface Builder can be extended to support custom views and objects that you or a third party defines. New views and objects are integrated into the application using plug-ins. A plug-in provides a list of objects to be incorporated into the library along with information about their attributes, outlets, actions, and initial configuration of those objects. The plug-in can also provide custom inspectors so that users can configure the object's attributes.

You can add plug-ins for third-party controls using the preferences window in Interface Builder. In many cases, however, you might not have to worry about doing so. System and third-party frameworks that define custom controls can include the Interface Builder plug-in for those controls inside their framework bundle. When you add such a framework to your Xcode project, Interface Builder automatically loads the associated plug-in. If your nib file is not associated with an Xcode project, you must load the plug-in manually using the preferences window.

In addition to using plug-ins to load third-party or custom controls, you can also use them to load custom configurations of standard controls. If you commonly use a set of nested views that must be configured in the same way every time, you can create a plug-in that contains those views configured the way you want them. At design time, you can then simply drag those views out of the library and drop them into your windows without having to do any further configuration. If the views are standard views, all you have to do is configure the library nib file of an Xcode plug-in project and build your plug-in. The default project template for plug-ins includes nearly all of the basic code required to load your plug-in. All you have to do is configure a nib file. For information about how to create a custom plug-in, see *Interface Builder Plug-In Programming Guide*.

Note: The plug-in format used by Interface Builder to integrate controls changed between versions 2.5 and 3.0 of the application. Plug-ins written for earlier versions of the application will not work in Interface Builder 3.0 and vice versa.

64-Bit Plug-Ins

Interface Builder 3.2 or later is a 64-bit application. Consequently, you should modify your plug-in to use 64-bit addressing. For general information about conversion to 64-bit addressing, see *64-Bit Transition Guide*. Both Interface Builder and ibtool can auto-relaunch in 32-bit mode to load 32-bit plug-ins.

Interface Builder Gesture Guide

Gestures for Making Connections

Using the Connections Panel

1. Control-click (or right-click) the source object and release the mouse button to display the connections panel.
2. Drag the circle next to the outlet or action you want to connect over the target object for the connection. (Hovering over an object causes it to reveal its children.) The target object should highlight to indicate a connection is possible. If it does not highlight, the target object is not of the right type and cannot be connected.
3. Release the mouse button to create the connection.
4. If you are configuring the source object's sent action, Interface Builder displays a list of the target object's action methods. Select the desired action method from the list to finish the connection.

Using Mouse Drag

1. Control-click (or right-click) the source object of the connection and do not release the mouse button.
2. While holding the mouse button, drag to the target object.
3. Release the mouse button over the target. Inter face Builder displays a prospective list of actions and outlets.
4. Select the desired outlet (of the source object) or action (of the target object) to create the connection.

Using the Connections Inspector

1. Select the source object and open the connections inspector (Command-5).
2. Follow steps 2-4 in "Using the Connections Panel" (page 169) to create the connection.

Modifier Keys

Modifier key	Description
Command	When you hold down this key, single-clicking an object toggles its selected state. You can use this modifier key to add objects to the current selection one at a time. You can also toggle the resize behavior for windows by holding down this key; see "Design-Time Resizing Modes for Windows" (page 95).
Option	After selecting an object, pressing this key and moving the mouse over a different object displays alignment information for that object relative to the selected object. Pressing the Option key during drag-and-drop operations copies the selected objects instead of moving them.
Shift	When pressed, single-clicking an object adds that object to the current selection. Pressing this key while resizing an object maintains the object's aspect ratio.
Control	When held down, clicking a Cocoa object displays its connections panel. Holding down the Control key is equivalent to right-clicking the object and the two behaviors have the same end result.

Inspector Gestures

Action	Gesture
Display inspector window	Shift-Command-I
Display attributes inspector	Command-1
Display effects inspector	Command-2
Display size inspector	Command-3
Display bindings inspector	Command-4
Display connections inspector	Command-5
Display identity inspector	Command-6

Library Gestures

Action	Gesture
Display Library window	Command-Shift-L
Add selected item to active workspace	Return or Enter

Workspace Gestures

Action	Gesture
Location	
Move selection by 1 pixel	Arrow (Left, Up, Right, Down)
Move selection by 5 pixels	Shift-Arrow
Size	
Resize selection to fit	Command-=
Guides	
Display position guides	Option (with selection)
Display layout rectangles	Command-L
Add horizontal guide	Command-Shift-_
Add vertical guide	Command-Shift-|
Navigation	
Reveal selection in document window	Command-Option-Up Arrow
Reveal selection in Library window	Command-Option-Right Arrow
Display in a popup menu all of the objects currently under the mouse	Shift-Control-Click

Document Window Gestures

Action	Gesture
Display icon view	Command-Option-1
Display list view	Command-Option-2
Display column view	Command-Option-3
Reveal selection in workspace	Command-Option-Down Arrow
Reveal selection class in Library window	Command-Option-Double Click

Document Window and Workspace Gestures

Action	Gesture
Activation	
Activate document window	Command-0
Activate document info window	Command-Option-I
Selection	
Select parent object	Command-Control-Up Arrow
Select child object	Command-Control-Down Arrow
Select next sibling object	Command-Control-Right Arrow
Select previous sibling object	Command-Control-Left Arrow
Editing	
Duplicate selection	Command-D
	Option-Mouse Drag
Navigation	
Display declaration for selection	Command-/
	Command-Double Click
Display documentation for selection	Command-Option-/
	Option-Double Click

Glossary

action A connection that involves the sending of a message from one object to another when a certain user action occurs. For example, when a user presses a button, the button object calls the action method of its target object to notify that object that the action occurred.

autosizing behavior A mechanism that automatically adjusts the size and position of views during resize operations based on a set of options. See also spring and strut.

automatic guide An alignment tool that shows the spacing required to meet the appropriate interface guidelines for the target platform. This type of guide appears and disappears automatically.

binding A two-way connection between the objects of your data model and the views of your interface. Cocoa bindings provide automatic synchronization between your data objects and the views displaying information about those objects.

connections panel A panel that appears as needed to display the connection status of outlets and actions.

content view In iOS applications, the portion of an iOS window that displays the application's custom content. Each content view may be represented by one or more actual views and typically presents a single screen's worth of application content. In Mac OS X applications, it is the view object that acts as the root for all other views in the window.

controller object An object that manages the interactions between an application's data objects and the objects that display that data.

Core Data A technology for managing structured data in your application. The data model of a Core Data application is built on a schema that defines one or more entities and their properties and the relationships between those entities. At runtime, Core Data manages the data for those entities using a database or other structured form of data store.

custom guide An alignment tool that is placed by the user in order to align objects to an arbitrary location on the design surface.

design surface The content area of a window object. This area is where you drop views and other visual objects and is also where you manipulate those objects directly.

dynamic guide An alignment tool that provides information about the position of a view relative to other objects on the design surface. Dynamic guides appear only when the Option key is held down.

event A type of connection that is specific to iOS applications. Events represent different phases of user interaction for a control. Each event connection can also have multiple assigned target objects, each with its own distinct action message.

File's Owner The runtime object that manages the contents of a nib file. The File's Owner is typically a controller object that maintains pointers to key objects in a nib file and responds to user interactions with those objects.

First Responder The object given the first opportunity to respond to events. The First Responder object is determined dynamically at runtime based on the several conditions, including which view is selected or has focus and which view is willing to accept certain types of events. If the First Responder does not handle an event, it passes the event to other objects in the responder chain.

interface object An object in a nib file that is created for your application at load time. Interface objects can consist of both visual objects (such as windows, views, and menus) and non-visual objects (such as controllers).

nib document The runtime representation of a nib file. Nib documents are the primary document type of the Interface Builder application. The on-disk representation of a nib document is a nib file.

nib file An on-disk collection of resource data. Nib files contain binary data representing one or more resources that you want to load into your application at runtime.

outlet A pointer to another object that can be set in Interface Builder. Applications use outlets to store references to objects in nib files.

placeholder object A placeholder for an object that is specified at runtime. Placeholder objects act as stand-ins for objects that are not available at design time. Instead, such objects are created by a running application and connected to the objects in a nib file when that nib file is loaded. Cocoa nib files use placeholder objects to represent the owner of a nib file's contents, the application object itself, and the first object to respond to events.

resource A generic term for structured data that is used at runtime to simplify the creation of some complex feature. Nib files store window resources, view resources, menu resources, and all the relationships between those objects and other objects in your application. When a nib file is loaded, its resources are turned into actual objects that can be used exactly as if they'd been created programmatically.

Responder chain The set of objects responsible for handling events in a window.

spring An element in the Size pane of the inspector window that controls the autosizing behavior of a view or control. When a spring is present, the width or height of your view grows and shrinks proportionally to its parent view. When no spring is present, the width or height of your view remains fixed.

strut An element in the Size pane of the inspector window that controls the autosizing behavior of a view or control. When a strut is present, the distance between the edge of a view to its parent view remains fixed. When absent, the distance grows and shrinks as the size of the views change.

xib file An XML–based version of a nib file. Xib files are the preferred format to use during development of your application. At build-time, they are compiled into nib files so that they can be deployed in your application bundle.

Document Revision History

This table describes the changes to *Interface Builder User Guide*.

Date	Notes
2010-07-12	Updated iOS terminology.
2010-03-25	Added information about the IBOutletCollection keyword and about making multiple connections in Interface Builder.
2010-03-24	Added information about iPad support.
2009-11-23	Made minor editorial corrections.
2009-10-19	Added information about user-defined runtime attributes.
	Replaced the term proxy object with placeholder object.
2009-08-28	Updated with new information about Interface Builder features.
	Added the chapter "Interface Builder Quick Start" (page 17) for new users.
	Described the new feature "Classes" (page 37) in the Library window.
	Added the appendix "Interface Builder Gesture Guide" (page 169).
2008-06-26	Updated to incorporate information about nib files in iOS.
2007-10-31	New document that describes how to use the Interface Builder application.

Document Revision History