8

Interface Builder

Library:	None, this API is defined by the Interface Builder application
Header File Directory:	/NextDeveloper/Headers/apps
Import:	apps/InterfaceBuilder.h

Introduction

This chapter describes the application programming interface that lets you build custom palettes, inspectors, and editors for Interface Builder.

Interface Builder gives you direct access to the majority of the objects defined in NEXTSTEP. Adding a Text object or a DBTableView object to your applicationÐobjects that represent years of programming and testing effortÐis as easy as dragging the object from Interface Builder's Palette window into your application's window. By creating a custom palette containing objects of your own design, you and other developers can manipulate these objects as easily as you do the ones in Interface Builder's standard palettes.

Using the facilities described in this chapter, you can easily create a palette that contains one or more objects of your own design. These objects can be of various types:

Type	Instantiation
View objects	Can be dragged into one of the application's standard windows.
MenuCell objects	Can be dragged into one of the application's menus.
Window objects	Can be dragged into the workspace.
Other non-View objects	Can be dragged into Interface Builder's File window.

The API described here also lets you provide inspectors for any custom object. There are four kinds of inspectors: Attributes, Connections, Size, and Help. The most common inspector to implement is the Attributes inspector, which lets the user set the custom object's unique features. For example, if you define a custom button object that sends a message repeatedly when it is pressed, the Attributes inspector could let the user set the repeat rate. Objects with special connection requirements (like those in the Database Kit) can provide their own Connection inspectors. The Size and Help inspectors are rarely overridden since they are appropriate for most types of objects.

If you need to provide the user with a more sophisticated system for interacting with your custom objects, you can implement an *editor* using the API described in this chapter. Whereas an inspector borrows one of Interface Builder's windows for its display, an editor provides its own window. The size of this window is not constrained as is the inspector window. Since each object can have its

own editor, there can be multiple editor windows on the screen at once, making ªcopy and pasteº and ªdrag and dropº interactions possible between editor windows. If the edited object contains other objects, the editor can open subeditors to let the user interact with the contained objects.

The DBModule object available from the Database Kit palette (in **/NextDeveloper/Palettes/DatabaseKit.palette**) provides an example of the use of an editor. If you drag a DBModule object into the File window and then double-click it, the editor opens its window.

To provide a better context for the discussion of the programming interface that makes custom palettes, inspectors, and editors possible, the next section gives a broad overview of Interface Builder's design.

Interface Builder's Design

You use Interface Builder to assemble and interconnect your application's objects. You start the process by creating a new document (or, more likely, by modifying the default document provided by Project Builder). When you save the document, it's represented by a file package having a name ending in ª.nibº. What's in this document or the nib file that represents it?

An Interface Builder document contains:

- An object hierarchy
- References to custom classes
- Connection information

Within Interface Builder, these components are managed by a *document object*. This object is of a private class, but can be queried and updated through the methods declared in the IBDocuments protocol.

The Object Hierarchy

A document object stores and maintains an object hierarchy. At the top of the hierarchy is the File's owner objectÐthe object that's represented in the top-left portion of the File window. This is actually a proxy object, since the actual object that owns the interface will exist outside of the nib file. When a user adds an object to the interface project, it becomes part of the document by being attached to some other objectÐthe *parent* objectÐin the object hierarchy. (In this hierarchy, a parent object may have many children, but each child can have only one parent object.) An object must be part of this hierarchy for it to be archived in the nib file.

Interface Builder declares and implements several methods as a category of Object (see the Object Additions specification) so that it can query any object in the hierarchy for crucial information. For example, each object can identify its various inspectors and its editor since it inherits these methods:

```
getInspectorClassName
getConnectInspectorClassName
getSizeInspectorClassName
getHelpInspectorClassName
getEditorClassName
```

When you define a class for a custom palette object, you can override any of these methods to provide your own inspector or editor.

Class References

Often, the object you want instantiated when your application runs is not available to Interface Builder either from its own library of objects or from any palette that has been dynamically loaded.

For these cases, Interface Builder provides a proxy object such as the CustomView object in the Basic Views palette. When you drag a CustomView into your application, you are in fact adding this proxy object to the document's object hierarchy. When the resulting nib file is loaded within a running application, the proxy object is unarchived and queried to determine the identity of the class that the proxy represents. Then, an instance of this custom class is created (through the facilities of the **alloc** and **init** messages), and the proxy is freed.

Note that this distinction between objects that are unarchived and objects that are represented by proxies has important consequences. An object that's unarchived can receive **awake** and **finishUnarchiving** messages, but won't receive an **init** message. On the other hand, an object that's represented by a proxy object in the nib file will only receive an **init** messageÐit won't receive an **awake** or **finishUnarchiving** message.

Connection Information

An Interface Builder document also contains information about how objects within the object hierarchy are interconnected. This connection information is embodied in objects that conform to the IBConnectors protocol. Each connector object stores information about a connection between one source object and one destination object. Interface Builder's Connections inspector is the interface to a document's connector objects. Each time you connect a source object with a destination object, you are creating another connection object.

When you save the document, connector objects are archived in the nib file along with the objects they interconnect. When an application loads the nib file, the objects from the object hierarchy are unarchived, proxy objects are replaced with the appropriate instances, and connection objects are unarchived. Interface Builder then sends each connection object an **establishConnection** message, giving it an opportunity to connect its source and destination as it deems appropriate. For example, the standard connection object that Interface Builder provides (again, of an unspecified class) stores the identity of the source object's outlet variable and the destination object's action method, if any. So, when such a connector object receives an **establishConnection** message, it sets the source object's outlet to the destination object andÐif the source object's outlet is named ªtargetºÐit sets the source's action to the destination's action method.

In most cases, Interface Builder's standard connection objects will be sufficient for your needs. However, you can create a Connection inspector and connection objects of your own, and through the methods declared in the IBDocuments protocol, you can have these connection objects archived in the nib file. Also, note that since connection objects are archived in the nib file, and since they all receive an **establishConnection** message when the nib file is loaded, they provide a convenient mechanism for storing any sort of information, not just connection information.

Interface Builder's Programming Interface

The API that Interface Builder defines is organized as two class definitions, several protocols, and several methods that are added, through the use of categories, to the definitions of the Object and View classes. The function of these components is summarized in the following tables.

Classes

Interface Builder uses these two class definitions as links to your custom palette and to inspectors. It's through the methods defined in these classes that Interface Builder locates and loads the user-interface objects that appear in the custom palette and in the inspector for a custom object.

IBPalette	This class is provided as the owner of palette's interface. If your custom palette includes only View objects, there's no

	reason to subclass IBPalette. If the objects that appear in the palette represent MenuCells, Windows, or other non-View objects, you'll have to create a subclass of IBPalette to associate the images in the Palette window with the real objects you intend to have instantiated.
IBInspector	This is the abstract superclass for inspectors. Your inspector provides Interface Builder with the controls to be loaded into the Inspector panel when the user attempts to inspect the custom object. The inspector also interprets the user's actions on these controls as commands to modify the custom object's state.

Protocols

These protocols define the ways your dynamically loaded palette module can communicate with Interface Builder (the IB and IBDocuments protocols) and the ways Interface Builder can communicate with objects in your module (the remaining protocols).

IB	This protocol gives you access to global information: the object that represents the active document, whether Interface Builder is in test mode, the source and destination objects of a connection, and so on.
IBDocuments	This protocol defines the programming interface to a document object in Interface Builder. Through this interface, you can add and remove objects from the document's object hierarchy, add or remove a connector object, and set the active editor.
IBInspectors	This protocol declares the methods that all inspector objects must have: **ok:**, **revert:**, and **wantsButtons**.
IBEditors	This protocol declares the methods through which Interface Builder can interact with an editor object. Interface Builder invokes these methods to make the editor's selected object visible; to copy, paste or delete the selection; and to open an close subeditors, among other things.
IBDocumentControllers	This protocol declares the notification methods Interface Builder can use to inform an object in your module about the state of the documentÐthat it has been loaded or that it's about to be saved. You use the IB protocol to register an object as a document controller.
IBSelectionOwners	Editor objects conform to this protocol, which declares methods for counting the number of objects in the selection and for filling a List object with the objects in the selection.
IBConnectors	This protocol declares the methods that connector objects must implement. These include methods for identifying the source and destination of a connection and for establishing the connection between these objects.

Other Programming Interfaces

Through the use of categories, Interface Builder adds methods to the Object and View classes.

Object Additions	Interface Builder uses these methods to discover the various inspectors for the selected object. Default inspectors and editors are provided for all objects.

View Additions	These methods let custom View objects control how they are resized and redrawn.

Creating a Custom Palette

The process of creating a custom palette is most easily explained by example. See Chapter 18, ªBuilding a Custom Paletteº in the *NEXTSTEP Development Tools and Techniques* manual for such an example. (This information is also available on-line in **/NextLibrary/Documentation/NextDev/DevTools/18_CustomPalette**.)