NXApp, Spring 1994 (Volume 1, Issue 2).

Creating Advanced Interface Builder Palettes

written by **Thomas Burkholder**

Part of what makes Interface Builder unique is that it allows developers to extend its capabilities. However, many techniques for building tools to extend it are obscure, or at least tricky. In particular, creating Connections Inspectors and editors takes some practice; fortunately, the results are well worth the effort.

INTRODUCTION

Everyone's seen itÐthe classic, whizzy Interface Builder (IB) demo that leaves crowds of programmers gasping for oxygen and scrambling for their checkbooks. Yet, some competing products look nearly as good and as smooth. Rhetoric and flashy demos aside, what makes IB unique? The difference is simple: extensibility. Interface Builder takes advantage of the capabilities of Objective C objects, particularly dynamic binding and loading. The ability to archive objects and load bundles is what sets IB apart.

Extending IB by writing palettes is beneficial for two principal reasons. First, it improves your development efficiency. After you've put the object in a palette, you write fewer lines of code and can lay out the interface objects instead of programming them. This means you spend a lot less time fiddling with your interface code. IB provides this advantage even if you don't write any palettes; however, if you do write palettes, these productivity gains extend to them.

The techniques and objects discussed in this article are available from NeXTanswers™ as document #1256, ªTTools.º It's a good idea to have the palette on-hand while you read the article, because everything in the article is implemented in the palette.

The second major reason to put objects in a palette is to reduce other developers' learning times. A palette is a perfect vehicle for ObjectWare, allowing you to package objects neatly without

requiring developers to learn much about the objects before using them. In IB, your object is treated very much like other objects; any developer can learn the general functionality and basic interface to your object with a minimal amount of effort. This advantage is as important for large organizations as it is for third-party developers: Even an experienced developer can't knowÐand usually doesn't want to knowÐevery detail of the interface of every object.

This article provides an overview of what IB can do and demystifies some of the more obscure IB arcana, particularly those involving Connections Inspectors and editors. It also provides guidelines for producing useful and powerful palette objects. If you're unfamiliar with IB palettes, you would be best off to go through the tutorial in the *NEXTSTEP Concepts* book before trying the examples in this article.

INTERFACE BUILDER BACKGROUND

To understand and build your own palettes, you first need to understand how Interface Builder works and how palettes work with IB. To help out, this section reviews some IB basics. (If you're familiar with the internals of IB, you can skip to ªExtensible Elements of Interface Builder.º)

IB as an object archiver

IB is document-oriented, as are most NEXTSTEP applications. This means that its primary function is to load and save objects conforming to the IBDocument protocol. In IB's case, the documents are nib files. The on-screen representation of a nib file is the window that initially appears in the lower-left corner of the workspace, titled with the name of the file.

The first key to understanding IB is understanding what a nib file is. IB's nib files are collections of archived objects, where each archived object is a serialized version of an object. For example, a TextField can be translated into a series of bytes that can be stored in a nib file. Later, the nib file can be loaded, and the object unarchived to its former state and used.

In Interface Builder, when you drag a TextField onto a window, IB creates an actual TextField object; when you have finished making changes to the TextField, IB saves the object in the nib. The advantage of this is that when you run the application, the TextField is unarchived and set to exactly the state you left it in the nibÐin the same location in its parent view, with the same text, on the same background, in the same font. This sounds simple, but similar applications in other environments don't use this technique. Instead, they typically generate code that must then be linked into an appÐtheir results are only as good as their code generators.

Understanding Interface Builder as an object archiver is crucial. The entire extensibility of Interface Builder and all of its benefits spring directly from this one concept.

Elements of nib files

A nib document can be subdivided. The subdivisions are clearly delineated by the four suitcases labeled Objects, Images, Sounds, and Classes. The most important of these, for our purposes, is Objects. The Objects suitcase provides a sort of summary overview of the objects in the nib file. A number of standard and specialized objects are provided for you, like the First Responder, File's Owner, and, in the case of application nibs, the Main Menu object. Also, any window or custom objects that you instantiate go in this suitcase.

In contrast, when you instantiate a menu cell by dragging it into a menu, that object is owned by the Main Menu object, and therefore isn't represented in the Objects suitcase. The same is true of any View, such as a TextField, Button, or Matrix. These objects are contained by the windows onto which they're placed and are represented in the Objects suitcase by the containing windows.

The take-home point here is that every object saved in a nib file is represented in the Objects suitcase, directly or indirectly. *Everything else in IB is just a tool for changing those objects.* For instance, an Inspector is just a thin interface to the selected object's instance variables. When you edit values in the Inspector, you alter the state of those instance variables. Similarly, when you double-click a Button in a window and edit its title in place, you're really using an editor to change an instance variable of the Button. In a sense, the editor is really just another sort of Inspector.

EXTENSIBLE ELEMENTS OF INTERFACE BUILDER

Of course, the wonderful archival framework that IB provides would be fairly limited if it didn't offer you a way to extend it. This is why Interface Builder provides Application Programmer Interfaces (APIs) for you to add your own interface objects, Inspectors, editors, and even general-purpose features that the designers of IB might never have intended.

Ordinary objects and Inspectors

The usual way to create an IB palette is to write the code for an object in a palette and write an Attributes Inspector for it if necessary. This is appropriate in the most common situation: You simply want to allow people to use and configure an object you designed. For an example, see the Timer subproject in TTools.

To find out how to implement palette objects and Attributes Inspectors, see *NEXTSTEP Development Tools and Techniques*. It's helpful to understand the basic concepts before attempting to use the more complex functionality described in this article.

F0.tiff ,

Figure 1: *The Attributes Inspector for the Timer subproject of TTools*

The object you provide can be a subclass of any of five classes: Window, View, MenuCell, Menu, and ActionCell. Or, if it fits into none of these categories, you can treat it generically and say it's a direct or indirect subclass of the Object class. You use this classification in the implementation of your IBPalette subclass to tell Interface Builder where to drop instances of your objectÐinto the Objects suitcase, on a window, in a menu, or on the screen itself. You do this by sending the message **[self associateObject: type: withView:]** in the palette subclass's **finishInstantiate** method. (See the IBPalette class description for more information.)

An important note about objects in palettes: Make sure they correctly implement the archival methods **read:** and **write:**. Otherwise, Interface Builder won't be able to properly duplicate the prototypeº instance you provide.

Once you've provided an object, you'll typically write an Attributes Inspector for it. To tell Interface Builder which class to instantiate as your object's Inspector, have the object's class implement the **getInspectorClassName** method. See Figure 1 for an example.

Creating custom Connections Inspectors

In addition to Attributes Inspectors, you can provide Connections, Help, and Size Inspectors. Much of the functionality of these other Inspectors is based on the ordinary IBInspector functionality. This article discusses only Connections Inspectors, because they are, after Attributes Inspectors, the most useful extensions to objects in palettes.

Connections Inspectors, properly used, can add a lot of value to an object in a palette. However, you should implement a custom Connections Inspector only when the standard, IB-provided Inspector is too limited for the connection paradigm that best suits your object.

Example: SwitchView

The SwitchView class is a good example of a class that needs a custom Connections Inspector. SwitchView displays a single user-selected view from a list of views. One of the instance variables in the SwitchView class is a list of the views that can be swapped into the content area of the SwitchView; the instance variable is called *views*. The trick is to find a way for a developer to attach different views as items in the *views* list.

One approach, inelegant but functional, is simply to provide 20 or so outlets and let the developer hook them up in the ordinary way. Of course, this doesn't work if the developer wants to hook up more views than you planned.

A better solution is to write a Connections Inspector for SwitchView that allows the developer to connect items to the SwitchView. Each view that the SwitchView is connected to is actually added to the *views* list. To create this kind of Inspector, you must create an IBConnector-conforming class to manage the relationships between the SwitchView and the objects it's connected to and a Connections Inspector class to allow you to add, modify, and delete these connectors. Figure 2 shows this Inspector.

Figure 2: *The SwitchView's Connections Inspector*

Telling the object to use a different Connections Inspector

To use the new custom Connections Inspector, SwitchView must respond to the method **getConnectInspectorClassName** by returning ªSwitchViewConnectInspectorº, the name of its Connections Inspector class. It's a good idea to put **get...InspectorClassName** methods like this in a category of the object rather than in the object itself, to keep the IB-specific code separate from the more general object.

Creating the connector object

A *connector object* is ªglueº code that binds source and destination objects in customized ways. A connector can be an instance of any class but it must adopt the IBConnectors protocol. Among other things, it must implement archival messages so that IB can archive the connector in the nib file. The connector must also implement the source and destination methods in the protocolÐIB uses these methods in order to draw connection lines on the screen, for instance. The Switchview example uses **setSource:** and **setDestination:** methods as well, to properly set up the connector.

A connector object is very useful, because it's archived in the nib file. The connector's **nibInstantiate** method is called after all of the source and destination objects are unarchived so that the connector can establish the connection properly. For a target/action sort of connection, the connector should probably send messages like **[source setTarget:***destination***]** and **[source setAction:***someAction***]**, assuming the action selector was also stored in the connector. In the SwitchView example, the destination of the connection must be added to the *views* instance variable in the source object, so SwitchViewConnector's **nibInstantiate** method simply sends the message **[[source views] insertObjectAt:0]**.

Because the connector object must be archived in the nib, it must be linked into the application, usually as part of a library of the same name as the palette. If the connector hasn't been linked in, the nib isn't loadable. Even so, it's not necessary, and probably not advisable, to provide the interfaces to any connector objects you create, since they're never used after the nib is loaded.

Implementing the SwitchView Connections Inspector

The SwitchViewConnectInspector class, like any Inspector, needs to be a subclass of IBInspector. Like other Inspectors, it should load its interface in its **init** method. The protocol the Inspector must respond to is the same as for other Inspectors as wellÐit includes the methods **init**, **ok:**, and **revert:**. The differences between a Connections Inspector and other Inspectors lie in its implementation itself.

For example, the Connections Inspector's **revert:** method should get the list of connectors whose source is the selected SwitchView object. IB keeps a list of connectors and provides protocol methods like **listConnectors:forSource:** for this purpose. Also, the IBInspector class provides you with the *object* instance variable, which is the currently selected object. So, **revert:** can get the list of connectors by sending a message like this:

```
[[NXApp activeDocument] listConnectors:connectorList forSource:object];
```

The connector objects are then displayed in a browser on the Connections Inspector's interface.

Next, the **revert:** implementation should check the result of **[NXApp isConnecting]** to see if the developer activated the Inspector by Control-dragging a connection between the SwitchView and another object. If so, **revert:** should select the connector represented in the browser whose destination matches the object returned by **[NXApp connectDestination]** and set the Connect button on the Connections Inspector interface to Disconnect. If the returned object doesn't match any of the destinations of the connectors in the list, then the developer has initiated a new

connection; **revert:** should set the Connect button to Connect. If **[NXApp isConnecting]** tested false, then **revert:** should disable the Connect button.

The Connect button on the Inspector should be hooked up to the **ok:** action. The Inspector's **ok:** method should create a new connector object; it can get the source and destination of the connection by sending **[NXApp connectDestination]** and **[NXApp connectSource]**. Then the **ok:** method should add this connector object to IB's list of connectors, via the **[[NXApp activeDocument] addConnector:]** message. Once the **ok:** method refreshes the list of connectorsÐessentially invoking **revert:**Ðthe method is complete.

Editors

An editor in IB is a class responsible for visual modification of a selected object. This is different from an Inspector, which generally provides a veneer of interface over the selected object's instance variables. The primary purpose of editors is to provide a natural user interface to the object's state, while the purpose of an Inspector is to provide a more complete interface to the state. For instance, the TextField editor allows you to double-click on a TextField and change the **stringValue**. It doesn't, however, allow you to set any other attributes, like the border style. The best way to think about this is that the editor should allow you to modify the contents of a selected object, while an Inspector should allow you to modify its state.

For example, when you edit a Button on a window in IB, what is actually happening is that IB is fooling you a bit. The Button's editor places a Text object directly on top of the Button you're editing, and takes it away when you're done. Similarly, when you double-click a DBModule object, the object's editor creates a window so you can drag connections to and from the abstract database.

The current Interface Builder API doesn't allow you to implement view editors, such as the one that allows a developer to directly edit a TextField. It does, however, enable you to write editors for anything that you've associated with the IBObjectPboardType in your palette object's **finishInstantiate** method; for example, you could create an object editor like the DBModule editor. Don't despair; even this kind of editor is surprisingly powerful.

Example: ListEditor

A simple example of an object editor is ListEditor. Being a container class, List is a prime candidate for an IBEditor. The ListEditor in TTools allows you to add and delete objects of

different classes and to cut and paste objects. It also lets you select individual objects in the list; the primary purpose of this is to allow you to delete the selected object and select an object you've just added. All the ordinary pasteboard operations apply to the selected object. In addition, IB can inspect the selected object. This is where IB's versatility really shines: A developer can inspect a selection in the custom editor just like any other selected object in IB.

The window ListEditor generates when a developer double-clicks a List object is shown in Figure 3. The Class and Instance radio buttons dictate whether to add class objects or instances. The browser immediately below is a class browser. The Show By field allows the user to enter a selector name; the editor uses the string result from this selector name to display the objects in the list.

See the IBEditors protocol for more on creating custom editors.

F3.tiff ,

Figure 3: *A ListEditor*

The objects themselves are displayed in the lower browser, by the result of the **name** method if the Show By field has not been changed. The Add button adds an instance of the class specified in the upper browser, or the class object itself, if the radio button dictating this choice is set to Class. The Remove button removes and frees the selected object. Like any other selection in IB, selected objects in the list can be inspected with the ordinary Attributes Inspector. Moreover, selected objects can be edited: A developer can add a List object to a List object, and still edit the nested List object. This can be very powerful.

Implementing ListEditor

Ok, now to the nitty-gritty part: How is it done?

To start, List needs a way to tell IB that it should use the custom editor. Therefore, List should have a category that implements the method **getEditorClassName** to return ªListEditorº.

The editor is initialized in much the same way as InspectorsÐthe intializer in this case, however, is IBEditor's **initWith:inDocument:** method. This method loads the nib section that contains the editor window with the editor object itself as the owner. In addition, this method should set up the browsers to their initial state.

The **free** method should close all subeditors and free any selection list. It should not free any of the objects that have been put in the selected List object.

ListEditor manages the selection in an internal list called *selection*. IB uses a number of methods on the editor to manage the selection. These methods are described by the IBSelectionOwners protocol. For ListEditor, they're simple to implement. IB sends the editor **getSelectionInto:** to get a list of selected objects. It might do this, for instance, if the developer using the list selects something in the editor and then tries to inspect itÐIB needs to know which object it's inspecting. Of course, one can inspect only a singular selection, so IB uses **selectionCount** to check the number of objects selected. In addition, IB requires the method **redrawSelection**, which redisplays the selection.

In addition to the selection-management methods, IB uses the methods **wantsSelection**, **selectObjects:**, and **makeSelectionVisible:**. These methods are selection *input* methods, in contrast to the *output* methods above. The **wantsSelection** method should return a boolean value indicating whether the editor can become the selection owner. (It's possible you might want to implement an editor without objects to select inside it.) IB uses the **selectObjects:** method when it wants the editor to select objects. In ListEditor, this method is mostly used by other methods, after they've changed the selection. The **makeSelectionVisible:** method applies mostly to View editors (which you can't implement), so for it the ListEditor just returns **self**.

IB also requires that you implement methods to cut, copy, and paste objects. It invokes these after it receives a cut, copy, or paste directiveÐthe methods are sensibly called **cutSelection**, **copySelection**, **pasteSelection**, and **acceptsTypeFrom:**. In the **copySelection** method, ListEditor merely acquires the NXGeneralPboard object and pastes the selection list in it, using **copyObjects:type:inPasteboard:** from the IBDocuments protocol. Other parts of TTools use the **[NXApp activeDocument]** message to acquire the document to send this message to; in this case it isn't necessary, because the ListEditor receives the document **id** when it's initialized and stores it in an instance variable. The pasteboard type to use in moving around the selected objects is IBObjectPboardType. IB sends the message **acceptsTypeFrom:** to decide whether to send the message **pasteSelection**Ðif the objects have no acceptable types, the **pasteSelection** method should generate a beep, as in any invalid pasting operation. In ListEditor, as long as one of the types passed in the argument is IBObjectPboardType, the paste is allowed.

A few methods manage opening and closing editors. ListEditor's **close** should send the message **[document editorDidClose:self for:list]** with the list as the edited object, and then should free itself. **openSubeditorFor:** is sent when the developer double-clicks on a selected object in the

lower browser. In fact, this method isn't invoked by IB, but by an action method hooked up to the doubleAction of the lower browser. **openSubeditorFor:** should have IB create a new editor with **editor = [document getEditor:YES for:anObject];**. Then it should activate the editor. In the case of Object editors, it's a good idea to move the window a bit at the end so that it doesn't completely obscure the editor it was opened from. Finally, **closeSubeditors** should simply iteratively acquire an editor if one exists for each item in the list, and send each the **close** message.

There are a few more small methods that IB requires as well:

window returns the window that the editor opens

editedObject returns the object that was initially passed to initWith:inDocument: (the currently selected object)

document returns the document that was passed in the intializer

orderFront orders the editor's window to the front

activate sets up the editor to become the selection owner again

activate always calls **[document setSelectionFrom:self]** to let IB know what objects the developer is selecting.

Clear as mud now, isn't it? The API for editors is a bit large, but overall very simple. One way to learn more about how editors work is to implement a stub editor, one that implements all of the necessary methods by popping up an alert panel with the name of the method and then returning an appropriate value. Play with it a bit in IB, and you will quickly find out when and why each message is sent.

PALETTE TIPS AND TRICKS

A few common problems and situations tend to arise in any challenging palette project. Here are some basic tips and tricks to help you deal with the most common conundrums.

Managing palettes with common symbols

A common problem in writing palettes arises when two palettes use the same library. Interface Builder won't load both palettes because they contain identical symbols, those found in the library common to both palettes. There are two usual ways to get around this; I also suggest a third,

which I think is more elegant and which is included in the Bundles subproject of TTools.

Don't import symbols

The first and easiest way around the problem is to use the **-u** linker option on one of the palettes so that it compiles as if the library were linked in, without actually importing the symbols. This flag is often used in creating relocatable objects, to use symbols that are linked into the application that will load the bundle. Of course, the first palette that you load must actually link the library in so that the symbols are available when the second palette is loaded. And, you must load the first palette before the second one. So, this solution creates a different problem in that it makes one palette dependent on another, which can adversely affect your palettes' portability.

Make a common bundle

The second way around the problem is to put all of the common symbols in a separate bundle and have each palette load that bundle if necessary before loading code that depends on the symbols. However, this makes it necessary to implement three bundles per palette: The palette itself is one bundle, the common code is in a second bundle, and a third bundle depends on the common code. Your palette won't load if it refers to symbols in the common bundle, because the common bundle won't yet have loaded. So any code depending on the common bundle must be loaded after the common bundle.

Make IB load your common bundles

The solution I prefer is similar to the first one, but more general. The first palette loaded should add the capability to load generic bundles into Interface Builder. It can do this by adding a menu item to Interface Builder's Tools menu. Set the target of the menu item to a private object that loads a Bundles window with user interface controls that allow the user to load bundles. Add a little bit of extra code so that the bundles that are loaded are remembered between runs of IB, and now instead of having many palettes with a confusing web of dependencies, you have palettes that use common code that depends on the bundles explicitly loaded into IB. Of course, these dependent palettes must be loaded after the palette that adds the bundle-loading capabilities, so in that sense, they all depend on one palette.

In general, adding menu items to IB by accessing the main menu or, for instance, adding a global object that listens for a DO connection is a dangerous thing. Make sure that what you add is a general feature, not a specific hack designed to make just a particular feature work. As a general rule, test palettes very thoroughly before releasing them; developers after all are trusting you not to crash Interface Builder or corrupt nib files. This rule is even more important when you provide

extensions to IB itselfÐthe potential for introducing bugs is *enormous*.

Providing custom cells

A common IB question is ªHow can I provide a custom cell that can be Alternate-dragged into a matrix?º This is a bit arcane. The first step is to provide a Control object that uses the custom cell. For example, consider a subclass of Slider called CircularSlider. The CircularSlider class should use the CircularSliderCell class, which is a subclass of SliderCell, just as Slider's cell is a SliderCell. (Try saying it five times fast.) The Application Kit™ documentation on the Control class shows how to implement **setCellClass:** and **initFrame:** so that the subclass of Control uses the correct cell class. The cell class that the Control subclass uses must be a subclass of SliderCell, TextFieldCell, MenuCell, or ButtonCellÐIB won't expand other cells into matrices.

ΣλιδερΧελλ.επσ ←

Figure 4: *A partial class hierarchy*

Now when a developer Alternate-drags a CircularSlider object, IB asks the CircularSlider for its cell, using the **cell** method inherited from Control. Then, IB creates a Matrix and uses the CircularSliderCell object as the Matrix's cell prototype. Every time the developer adds another row or column, in IB or from a method call, the Matrix uses the CircularSliderCell's **copy** method to create more instances; be sure that you've implemented **copyFromZone:** in CircularSliderCell so that this works properly.

When you implement an Inspector for CircularSlider, in the **revert:** and **ok:** methods you may want to determine whether the inherited instance variable *object* is a CircularSlider or a CircularSliderCell. Once IB can generate a Matrix of CircularSliderCells, it uses the CircularSlider Inspector to inspect the cell as well. If you've written your classes carefully, you can often make the Inspector work whether the cell or the control is selectedÐthis is a good measure of how correctly you've implemented your control and cell subclasses.

Another Inspector note: When you design your Inspector, make sure the window is shorter than the standard Inspector by at least the height of a default button. This is because the Matrix prototype Inspector uses OK and Revert buttons. Note as well that the Match prototype feature of the Matrix Inspector won't work for attributes that aren't inherited from one of the four ActionCell subclasses. However, if you change the prototype, new cells created by the Matrix are copied from the prototype.

Subclassing Matrix

The Matrix class is surprisingly flexible and can be tuned to a variety of purposes. Occasionally, however, you may want to add value by subclassing it. An example of a Matrix subclass is Ranker, a Matrix that allows you to Control-drag cells to reorder them.

To gain the true power of these extensions, a developer using your Matrix would probably like to use the subclass in IB, to lay out and test interfaces without having to compile and debug the application. Unfortunately, IB is single-minded about which Matrix class to use when one Alternate-drags a control: It always uses the Matrix class. But, you can make a palette with a Ranker like any other viewÐset it up to have a particular prototype, put it on a palette, and ªit just works.º Because Ranker inherits Matrix's editor, a developer can Alternate-drag it just like an ordinary Matrix.

However, this method is a little limited. In particular, Ranker has to start with a prototype cell, which can be configured but not replaced. Of course, you could provide a number of different instances of Ranker on a palette, one each perhaps for ButtonCell, TextField, and Slider. But what about CircularSliderCell? You can provide a Ranker instance for it too, but not for cells that someone else might provide in a palette. And anyway, there are an awful lot of instances of the same class on the palette now. What's needed is a dynamic way of specifying the cell prototype.

For Ranker, I elected to use a simple, brute-force method. When the Ranker is dropped on the window, it generates a panel that queries the user for a cell class. The code entry-point for this is the **finishUnarchiving** method. When the Ranker instance drops on a window, IB copies the version associated with the dragged view by using the Ranker's **read:** and **write:** methods. But when an object is unarchived, as it must be when IB invokes **read:** to finish copying it, one of the last methods invoked is **finishUnarchiving**. This method allows you to replace the object with another if necessary, but Ranker uses it for another purpose: It displays a modal panel containing a class browser. The modal loop ends when the developer selects an appropriate cell class; then, **finishUnarchiving** creates an instance of that cell class and makes it the Ranker's prototype. When **finishUnarchiving** returns control to IB, the matrix has a valid prototype of the class that the developer specified. The class browser is generated dynamically from the Objective C runtime system, so that when a new palette is loaded with a new cell class, we can make a Ranker containing those cells.

Using ªcoverº views

Suppose you're putting Ranker in a palette. The usual way to approach the task is to put a CustomView on the palette window and then use the CustomView Inspector to change the view's

class to Ranker. CustomViews are an exception in IB; they stand for other views that aren't real classes when you define the view, but are real classes at run time. When the palette is loaded, the CustomView is unarchived from the palette's nib file and, because it is a CustomView, the **loadNibSection:** code creates an instance of the CustomView's ªintendedº class, Ranker. It does this in the ordinary way, by sending **[[Ranker alloc] initFrame:]**.

This usually means that you can't lay out your palette very realistically in IB, because instead of a real Ranker you have this rectangle that says Ranker on it. And in Ranker's initFrame: class, you'll have to do some IB-specific things like setting up the Ranker in a certain way that isn't necessarily the way you'd want to initialize it in the general case. What this all boils down to is that using CustomViews to represent your object in IB tends to create problems in how you write the actual palette class.

The way around this is to use a cover view rather than a CustomView on your palette window. For example, for Ranker, it makes the most sense to use a Matrix object, because the two look identical. You can configure it to tasteÐin this case, it's three cells high and has the words ªRankerÐAn Extended Matrixº in the cellsÐand then connect it to an outlet of the palette object itself. In the example, this outlet is rankerCover.

Φ1.τιφφ ,
Figure 5: *Using a cover view in a palette*

In addition, the palette needs an outlet called rankerView, which must be connected to a CustomView that's marked as a Ranker object. You shouldn't leave this blank Ranker instance lying around on the main palette window; put it on another window that you never show. Make sure this window is nondeferred!

The final step is to put a line like the following in the palette object's **finishInstantiate** method:

```
[self associateObject:rankerView type:IBViewPboardType with:rankerCover];
```

This is in fact the way to make a palette of non-view objects in IB. View is the only class for which the message **[self associateObject: type: with:]** isn't required in a palette object's **finishInstantiate** method. But, even though it's not required, it's good to include anyway, because it makes layout of the palette window a lot cleaner. Even Interface Builder is built this way: In Interface Builder's **English.lproj/Menusnib**, the standard menus that IB provides are

actually Buttons on the palette itself. (Note that the icons on the Buttons are NXmenuArrow and NXmenuArrowH, some useful global images that don't show up in the Images suitcase of the nib.)

Using ªaccompanyingº objects

When you drag a Font or Format menu item from a palette, you automatically get a FontManager object in your Objects window, complete with target/action connections from the various items in the Font menu. Although no standard APIs that are really intended to do this, it can be implemented, with a little bit of trickery. The ShowMenus subproject of TTools gives an example.)

F7.tiff ,

Figure 6: *The Show Menus command in action, in the Draw application*

The example creates the menu item Show Menus. It expands all of your application's menus side-by-side so that you can see them, as in Figure 6Ða useful feature provided by many third-party applications. So when a developer drops the Show Menus MenuCell on an application's main menu, a manager object appears in the Objects suitcase and a connection is automatically made between the new MenuCell and the manager object.

Setting this up is intricate. Here are the steps:

1. Make a palette of a subclass of MenuCell.
2. Gain a code entry-point when the cell is dropped into its destination menu.
3. Create a manager object and cause it to appear in the object hierarchy.
4. Create a custom connector between the new MenuCell subclass and the manager object.

Although there's no reason to subclass MenuCell just to get a Show Menus title, I found it necessary to subclass it anyway, since I needed a code entry-point to set up the MenuCell's target and action. The subclass is ShowMenuCell. Recall that Ranker uses the **finishUnarchiving** method to get an entry point to when the Ranker object is dropped on a window; this is the same trick. The implementation of ShowMenuCell's **finishUnarchiving** method looks like this:

```
static id masterShowMenus;
```

```
- finishUnarchiving
{
   id conn;

   // generate a master ShowMenus object, if one doesn't exist
   if (!masterShowMenus) {
      masterShowMenus = [[ShowMenus alloc] init];
      [self installMasterShowMenus];
   }

   // When bug in isTestingInterface is fixed, remove the following lines.
   // See IBShowMenus.m for more info.
   if (![[NXApp activeDocument] objectIsMember:masterShowMenus])
      [self installMasterShowMenus];

   // Set up a connection
   conn = [[SMConnector alloc] init];
   [conn setSource:self];
   [conn setDestination:masterShowMenus];
   [[NXApp activeDocument] addConnector:conn];

   // Could set up and return an ordinary MenuCell, but IB barfs on
   // my connection (because it's expecting its own connector class-
   // this is really a bug in IB).
   return self;
}
```

Finally, I set up a connection between the MenuCell and the manager object. The connector object is very similar to the one described for custom Connections Inspectors, and simply sets a target and an action when it is invoked at nib-loading time.

If you can bend your mind around all that, you'll notice there's a rabbit to be pulled out of a hat: the **installMasterShowMenus** method.

```
- installMasterShowMenus
{
   // Make IB aware of it
   [[NXApp activeDocument] attachObject:masterShowMenus to:NULL];
```

```
    [[NXApp activeDocument] setName:"ShowMenus" for:masterShowMenus];

    // Hack to make it appear in the Objects window
    [NXApp displayConnectionBetween:masterShowMenus and:masterShowMenus];
    [NXApp stopConnecting];
    return self;
}
```

First, the method tells IB about the new object in the nib's collection of objects to archive, using **attachObject:to:**. Then, it sets up the name of the object so that the developer has some idea of what's been added to the Objects suitcase. Next, to make the object actually appear, **installMasterShowMenus** tells IB to display a connection that has the masterShowMenus object as both source and destination. It immediately calls the **stopConnecting** methodÐthis produces a visual flicker when the connection appears and vanishes. (This is unpleasant. If anyone finds a better way, please let me know.)

CONCLUSION

Tool-building in NEXTSTEP is an important but often neglected art. Objective C and the Application Kit together provide excellent, reusable building blocks for your application framework, but these building blocks alone aren't enough. State-of-the-art, reliably built, versatile development tools are as important for the object-oriented programmer as the equivalent tools are for a carpenter. With well-crafted windows, quality lumber, and good fastening hardware, a carpenter using only a handsaw, hammer, and nails could probably build you a mansion; however, it would take a few years. Provide the same carpenter with advanced power tools, and the same task could be completed in a fraction of the time, with equivalent or superior quality. In a similar way, Interface Builder is the power tool of NEXTSTEP; well-written IB palettes augment that tool to an even greater degree, increasing reusability and decreasing development time.

Thomas Burkholder is a member of the Developer Support Team. You can reach him by e-mail at **Thomas_Burkholder@next.com**. Please feel free to send him comments, suggestions, and bug reports regarding the TTools palette.

TARGET/ACTION PARADIGM

One of the most common questions about Interface Builder is, "How do I allow people to define a target and an action for my custom object?" People often discover that if their object is a subclass of Control, this behavior is automatic. The reason, however, is that Controls respond to an implicit protocol that I unofficially call the "TargetAction" protocol. The methods in this protocol are **target, setTarget:, action**, and **setAction**:. In fact, any real (not parsed) object that you implement that responds to these methods behaves as a target/action source in IB.***-TB***

USING THE SUPERCLASS' INSPECTOR

Generally, it's more useful to add functionality to existing objects like TextField and Matrix than it is to develop objects from scratch. Of course, once you've added this useful functionality to your subclass, you typically provide an Inspector that allows you to configure any new instance variables or perform special tasks associated with the new functionality. However, once you've implemented a new Inspector, you lose the features of the superclass Inspector. This can be a major loss, particularly when the superclass is Matrix, whose Inspector implements many features that are hard to reimplement.

There are a number of solutions to this, but this is perhaps the simplest:

```
- (const char *)getInspectorClassName
{
 NXEvent *ev;

 ev = [NXApp currentEvent];
 if (ev->flags&NX_ALTERNATEMASK) // the superclass's Inspector
  return [super getInspectorClassName];
  else // the real Inspector
 return "RankerInspector";
}
```

Tip: Put this method in a category, or you'll get a compiler warning because the Matrix header file does not show a **getInspectorClassName** method. It's a good idea to use categories for this sort of thing anyway, so that you can put IB-specific code in a file separate from the implementation of the object itself.***-TB***

AWAKEFROMNIB BUG

The **awakeFromNib** method is useful because it's invoked in your object after all connections to or from it are made-this means that you can depend on outlets, like those that point to what you connected them to in Interface Builder. In Test Interface mode, however, **awakeFromNib** isn't called. This typically means that your object isn't initialized the way you intended in Test Interface mode.

There are two workarounds. First, if your object is a View subclass, you can make the **drawSelf::** method detect

the uninitialized state and do something about it; this is what the SwitchView example does. This approach works for non-view objects as well, provided you can pinpoint a method that is called soon after the object is archived and provided you can detect the uninitialized state.

When you can't fulfill these conditions, then you have to do something a little more tricky. This second approach requires that you invoke the **perform:with:afterDelay:cancelPrevious:** method from inside your finishUnarchiving method, as follows:

```
[self perform:@selector(awakeFromNib:) with:self afterDelay:0 cancelPrevious:NO];
```

The **awakeFromNib:** method should invoke **awakeFromNib** and redisplay views that depend on the changed state.

The danger in using this approach, of course, is that **finishUnarchiving** is called whenever the object is unarchived, rather than just after the object is unarchived from a nib file. As a result, this method causes **awakeFromNib** to be called, for instance, when you drop your view on a Window, since IB uses archiving to copy the object. Because of this, it's important to make sure **awakeFromNib** won't cause any nasty side-effects if it's called outside of its regular context.

A better fix for this out-of-band behavior would seem to be to check the **isTestingInterface** result, so that the **awakeFromNib** method would be invoked only in Test Interface mode. However, **[NXApp isTestingInterface]** returns NO during unarchiving, so this last approach doesn't work.***-TB***

ISTESTINGINTERFACE BUG

[NXApp isTestingInterface] is useful when you want your object to behave differently in test mode than it does while it's being inspected and edited. You would usually use it when the method you are in can be invoked from either context.

However, there are two bugs in **isTestingInterface**: First, it returns NO when your object is being unarchived at the start of the Test Interface mode; second, it returns NO when your objects are being freed just after Test Interface mode terminates. Typically, then, you get useless results from this method when you call it from **finishUnarchiving** or free.

There are very few general workarounds for this. The best you can do is to make sure that an object's **finishUnarchiving** and free methods don't introduce bugs from either context.***-TB***

Next Article NeXTanswer #1638 **Discovering the DBTableView Object**
Table of contents http://www.next.com/HotNews/Journal/NXapp/Spring1994/ContentsSpring1994.html