Advice for the Appkit-lorn

The central relevation in most NeXT programmer's lives is their first demonstration of Interface Builder. It tends to stick with you, like that time in Junior High when you first slow danced with a person of the opposite sex. You had an immediate vision of previously unimagined possibilities. A world that had been full of unfocused angst suddenly seemed to be pretty livable after all.

Of course, shortly after that—usually later that night—the angst returned with a vengeance, along with a lot of other unbecoming emotions. So it goes with NeXT programming. It beats the alternative, but there's a lot of things you have to learn before it becomes a pleasant experience. Even then it can be trying at times. But those sublime experiences make it worth the effort.

So, Wally-to-Beaver-like, I'll offer a few words of advice on NeXT programming for those new to the experience. Some of it is going to be simple advice that most old hands take for granted. ("They probably don't appreciate noogies.") Some of it is going to be a little more philosophical. ("Women.") And some of it other people could take issue with. After all, I'm playing the role of Wally Cleaver here, not Ward.

Displaying Lists of Data

It's common to display a scrolling list of discrete data in your application. A panel might contain a scrolling list of all the people in a department, for example, and picking one person from the list would cause a photo, address, and phone number to be displayed in other fields. Displaying data in the text fields is pretty simple. So how do you go about implementing the scrolling list of data?

Let's reduce what we're looking at here to its essentials, so we don't get bogged down in a lot of peripheral issues. Let's assume that you want to display a scrolling list of short pieces of text. That's it. A screen shot of what we want to have is shown below:

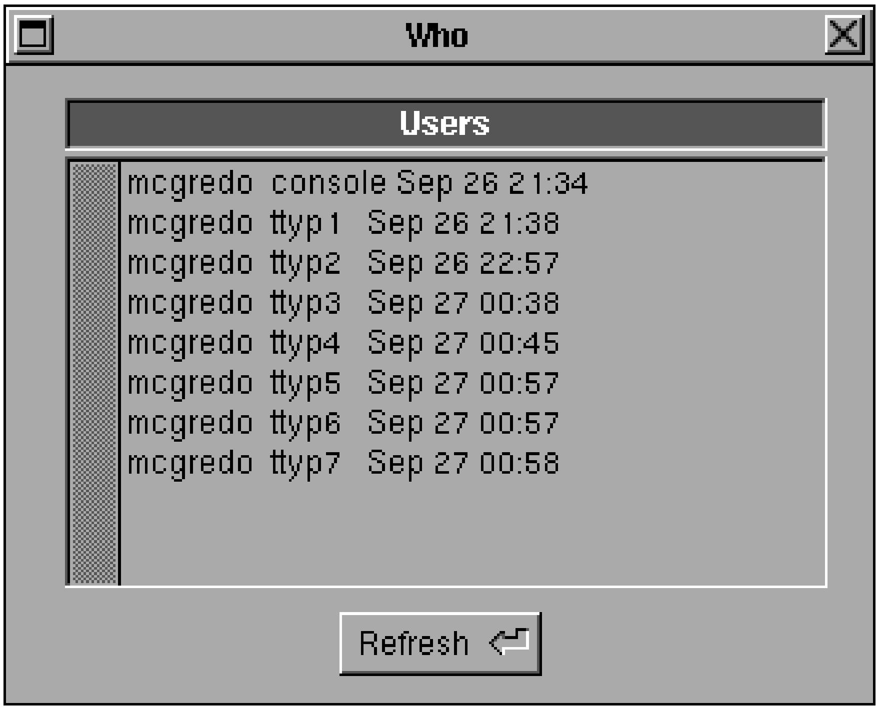

Figure 1. A window with scrolling text.

The text being shown here is just a list of who is logged onto the computer the program is running on. It's modeled closely on what the Unix who command does. (In fact, it is the Unix who command, gussied up with a GUI.) When the program is first run and whenever the user hits the "Refresh" button the data in the display is updated. It's not really important what the data is; we're just looking for something to display.

There are a few ways to do create the scrolling display. Among other strategies, you could group a Matrix in a ScrollView, then fill the Matrix with data. Or you could use a DBTableView--despite DBTableView's name and its residence in the DBKit classes, you can use it without a database. I'll do it yet another way, using the NXBrowser class. Not because it's better, but because it gives me a chance to demonstrate the important (and very nifty) Objective-C concept of delegation.

A delegate is a stand-in for the "real" object. The delegate performs some operation on behalf of the original object. To a newcomer to the concept, this probably seems like a weird idea. Why not just have the original object do it? Trust me, there are good reasons. We'll see some of them a little later, and explain some more about what delegation is.

Writing the Program

First some details on setting up the infrastructure for what we want. The source files for this example is on the Oregon State anonymous FTP server, cs.orst.edu, under the file name [...]. The example was written under NeXTSTEP 3.1.

Create a new project and open up the .nib file. From the palettes window of IB, pick the far-right icon, then drag over a browser object from the lower display. Then drag over a button, change the title, and drop a return-key icon onto it. You've just completed the interface portion of the project.

About Browsers

Browsers (properly called NXBrowsers) are intended to display hierarchical data. File systems are a good example of the sort of data that they're good at displaying. You can have directories inside of directories until finally you get to the "leaves," the individual files. This is exactly what's used for the browser mode view of the Workspace. A "real" browser can display several layers of data, but we'll keep it simple here and display just one. All our data will be leaf nodes—there will be no branch nodes and only one column in the browser.

Now we need some code to drive the interface and fill the browser with our simple-minded data set. Create a subclass of Object called TheDelegate. Add an instance variable, theBrowser. Parse it to make sure the instance variable shows up in IB, then instantiate the object. Control-drag a connection between the newly instantiated object and the NXBrowser object in the window, and make the connection to theBrowser.

Now the instance variable theBrowser of the object points to the real browser in the IB window. We want to make one other connection—we want the browser to have its delegate to be our newly created object of the class TheDelegate. Control-drag a connection from the browser object to the instantiated object, and hook up delegate so that it points to the instantiated object.

The under 3.0 and later releases, the browser is automatically sent the message loadColumnZero when the nib is loaded. (You need to send the message programatically under 2.x). This, in turn, causes other messages to be sent. One of those is -browser:fillMatrix:inColumn:.

Here's what the documentation supplied by NeXT has to say about this method:

> **browser:fillMatrix:inColumn:**
> - (int)**browser**:*sender*
> **fillMatrix**:*matrix*
> **inColumn**:(int)*column*
>
> Invoked by the NXBrowser to query a normal or lazy browser for the contents of column. This method should create NXBrowserCells by sending addRow or insertRowAt: messages to matrix. The NXBrowser will resize them to fit in the Matrix—you can't control the size of an NXBrowserCell. Returns the number of items in column.
>
> A normal delegate should create each NXBrowserCell and send them the messages setLoaded: and setLeaf:, and setEnabled: if necessary. A lazy delegate marks Cells as loaded only when they are about to be displayed; however, it may create and partially fill in

information (such as the title), saving only the time-consuming operations for an actual request to load an individual Cell.

If you implement this method, don't implement the delegate method browser:getNumRowsInColumn:.

See also: - **browser**:**loadCell**:**atRow**:**inColumn**: (delegate method)

This is typical of the documentation for the Appkit. It's got a bunch of code words in it you probably don't understand right now, and you can analyze it for hours without understanding any more. ("She said she hates me but she likes me, Wally. What does that mean?" "Gosh Beav, I guess it means she wants you to get lost. Or maybe not.") Later on, you learn to read between the lines and start to get the meaning of more of the code words. Things are clearer then. Sort of.

The browser:fillMatrix:inColumn: method fills one of the columns in the browser with data. Since we have only one column in this browser, that's all we need to fill the browser with what we want to display. The arguments to the method include what browser sent the message, the matrix that is going to be filled with data, and the column number of the browser we're working with.

Delegation

Now, back to delegation. The NXBrowser object does not implement this method—the NXBrowser object itself does not know how to load the information it displays. It relies instead on the delegate object, which can actually put data into the browser. In our case, the delegate is the object we created, an instance of the class TheDelegate. The NXBrowser instance realizes that it needs data to display when it receives a loadColumnZero message, and asks the delegate object to fill it with data.

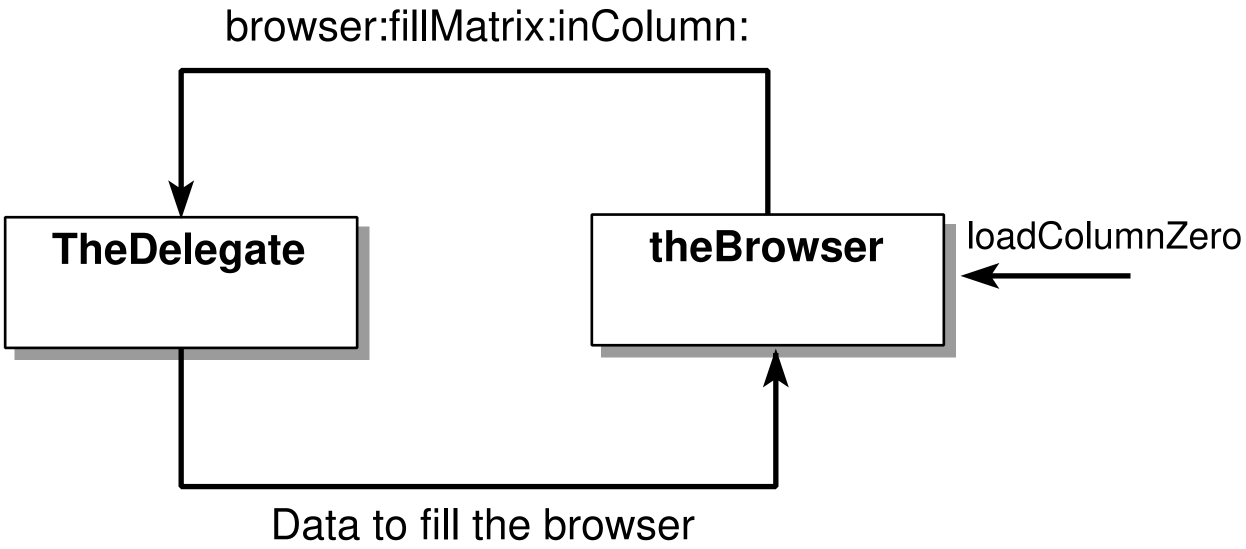

Figure 2. Delegate action for NXBrowser object

The object theBrowser receives a message, and in turn sends a delegate message to an instance of TheDelegate. That object in turn fills the browser with data.

Why do it this way instead of having the browser do it itself? In any large project, we'd have browsers used for maybe dozens of different purposes, each requiring its data to be loaded in a particular way. One option is to subclass NXBrowser for each use, with a new method implementation for each subclass. Soon we'd have dozens of subclasses of NXBrowser, all just a little bit different. The source files would be cluttering up your directory when you wanted to get to something. You'd have to come up with new names for every subclass (BrowserFileSystem, BrowserDOSFileSystem, BrowserMacFileSystem, BrowserOS360FileSystem, etc.) By using delegation, we can push the application implementation details out the the Browser object, and, we hope, make things a bit easier to understand. It lets NeXT distribute one general class that other people reuse with ease.

Newcomers to object oriented programming often assume that subclassing objects is the way to implement new behavior. 'Taint always true. Sometimes the it's better to use aggregation and cooperation among objects. This is probably one of those cases.

Objective-C uses run-time binding—the actual method called is not determined until the object actually asks for it during execution. This has some nice implications. One of them is that delegation is a lot easier to do. The NXBrowser object doesn't have to know what type of object its delegate is—it just sends the message, and the receiving object, whatever it is, looks up the right function to call. Run-time binding is one of the subtle things that makes NeXTSTEP what it is. Its use deeply influences the programming style and architecture of the Appkit. It's easier to build objects that cooperate with each other. In a C++ system, you'd probably have to use multiple inheritance to implement the NXBrowser features we described here. In effect, you'd have had to substitute lines of code for the lack of delegation and run-time binding.

I'd rather not write code I didn't have to.

Delegation is used in a lot of other places in the appkit. The Window classes have delegates. Whenever a window is closed a message can be forwarded to the delegate so cleanup operations can be performed, an alert box can be put up warning that user that the document hasn't been saved, and so on. The Application object has a delegate as well. Most programmers should not need to subclass Application at all; the necessary features can be entirely handled by the delegate object.

Often the delegate object will be the "owner" of the nib file. Think of a new document that is created from an application. A new nib file for the document is loaded, and the nib file is owned by an instance of a Document class. The Document object has instance variables that point to a browser in the document, the window created to hold the document, and so on. Both browsers and windows can have delegates, so the logical thing to do would be to make them point to the Document class instance. The Document instance would implement the delegate methods for loading the browser, checking to make sure the user has saved the document before closing, and other functions. This is shown in Figure 3.

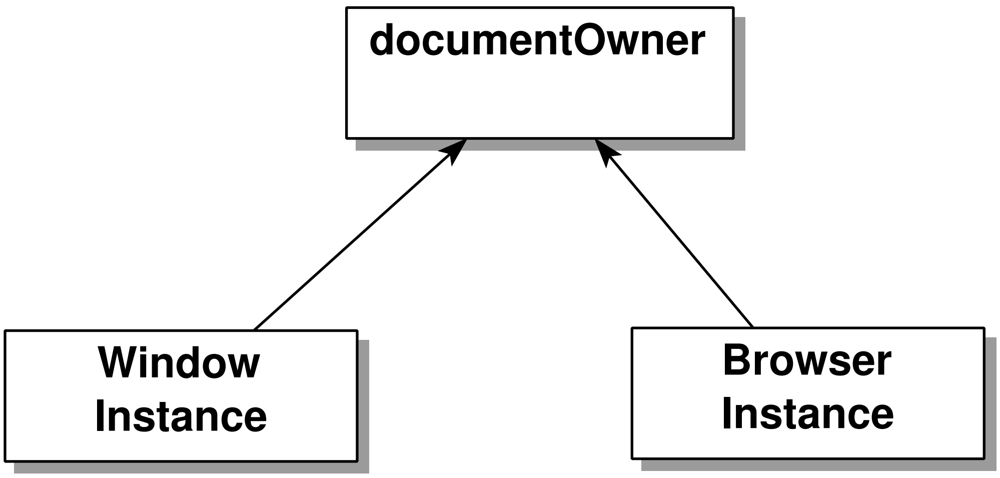

Figure 3. The browser and window have set their delegate to the documentOwner object.

The document owner would implement the browser:fillMatrix:inColumn: and windowWillClose: methods.

Finishing the Program

But back to the problem at hand. We want to fill the browser with data, so we'll implement a method in the delegate object to do that. Here it is:

```
- (int)browser          // RETURN: the number of cells loaded into the browser
        :sender            // INPUT:  the browser that sent us the message
  fillMatrix:matrix      // INPUT:  the matrix being added to
  inColumn:(int)column;  // INPUT:  which column of the browser is being loaded
{
/*------------------------------------------------------------------------------
The main method that loads the data.  This is the delegate method; When the
browser is asked to load itself, it calls the browser:fillMatrix:inColumn:
method in the delegate object, namely us.  We fill the matrix in the browser
by adding rows, then return a count of the number of rows returned.
------------------------------------------------------------------------------*/
Storage  *names;       // another ptr to the ivar storage object
int    i;              // generic counter
NXBrowserCell *aCell; // A cell we're adding

// get a pointer to the storage object that holds the names.  this is
// really just an alias to the ivar in the header file.

  names = [self getNames];

// Loop through all the names, adding them to the matrix.

        for(i = 0; i < [names count]; i++)
        {
          [matrix addRow];
           // Create a new default cell
         aCell = [matrix cellAt:i :0];       // get a pointer to it
         [aCell setStringValue:(char*)[names elementAt:i]];                    // set the text string

         [aCell setLeaf:YES];   // Needed to avoid the messy-looking icon on the right
        }
  return [names count];     // How many did we add?
}
```

That's it. We just add rows to the matrix, and set the text values of the cells inside the matrix. We'll want one other method, the action to be taken when the user clicks on the refresh button. This is really easy:

```
- refreshHit:sender;
{
```

```
    [theBrowser loadColumnZero];
    return self;
}
```

This just sends a message to the browser asking it to fill itself with data. The browser in turn queries the delegate object and fills itself.

From here, we can simply save the .h and .m files, parse the object, and connect up the button action by control-dragging a connection.

What We Learned

No Leave it to Beaver episode would be complete without a short moral before the credits roll.

Delegation is your friend. It is remarkably easy to make objects cooperate with each other in Objective-C due to run-time binding. You should be thinking in terms of Software ICs, in Brad Cox's term, reusable objects that cooperate with each other to get a job done. We also had a short tour of NXBrowsers—implementing a simple browser is very easy, and several other options to get the same result exist.

And maybe, just maybe, the beginners out there experienced the equivalent of a peck on the cheek from the experience that is NeXTSTEP.

About the Author

Don McGregor is a recovering OSU graduate student now living in Marin. There are a bunch of BMWs there, but he still misses the Bean and shuffling his feet through freshly fallen oak leaves on campus and watching the fog filter through the fir boughs after a long night of programming.