Q & A:

Q: How do I make an existing instance of a view class be the "cover" for the view class I want instantiated on my window? The documentation just says to use a Custom View with it's class assigned to be my custom view subclass. I can use a view to cover for an object or window, how do I do it with a view?

A: In much the same way you associate a "cover" view for an object that will be dragged into the Objects suitcase, you can do the same for View. The code in the "-finishInstantiate" method of your palette subclass should be something like:

```
[self associateObject:switchView
      type:IBViewPboardType with:switchViewCover];
```

Notice that the type is "IBViewPboardType". Obviously, the switchViewCover outlet should be hooked up to the view that you want to drag off the palette (I find buttons very useful for this) and the switchView outlet should be hooked up to the "CustomView" that's

been assigned to be your view subclass, usually on some off-screen window.

See:
;TToolsPalette/TToolsPalette.m;;¬TToolsPalette.m

Q: How do I implement an inspector in IB so that I don't lose my object's superclass inspector? I want to add functionality to an object in IB without having to re-write the inspector that modifies all the existing functionality.

A: There's a neat little four-line trick that does this. The gist of it is that in your -getInspectorClassName method, you return [super getInspectorClassName] when the alternate key is held down. Once this is implemented, you can access the superclass inspector by alt-clicking the object in question, or your new inspector by clicking on it in the usual fashion.

See:
;TToolsPalette/SwitchView.subproj/SwitchViewInspector.m;;¬SwitchViewInspector.m
;TToolsPalette/Ranker.subproj/Ranker.m;;¬Ranker.m

Q: Somehow, even though I implement a -getInspectorClassName method and an inspector for my Window (or Panel) subclass, I always seem to get the ordinary IB Window inspector. What's wrong?

A: Due to implementation details in IB, there are some objects that you can subclass and write inspectors for, and some you can't. Window, Panel, DBModule, and Menu are among them. You'll also notice that you will also be unable to write editors or customize the image used in the Objects suitcase (using the -getIBImage method) either. These are inherent limits of the current version of IB.

Q: I tried to implement an editor for my subclass of Button, but the -getEditorClassName method in my button subclass was never called. What's going on?

A: The current implementation of IB doesn't provide enough hooks to properly implement an editor for any view subclasses - the IBEditor functionality is intended to be used only with Objects that are dragged into the Objects suitcase. Although some view subclasses will be called with the -getEditorClassName method, most won't.

Q: How can I browse the class hierarchy?

A: There are two steps. First, an object-oriented way to get subclasses needs to be provided - this is in ;TToolsPalette/Utilities.subproj/ClassAdditions.m;;¬ClassAdttions.m, a category of Object that adds methods to provide you with a list of a receiving class's immediate child classes. The second step is to set up your browser's delegate. Because I found myself doing so many browser's delegates while building TTools, I decided to abstract a lot of the code into a new subclass of List, called ;TToolsPalette/Utilities.subproj/SortedList.m;;¬SortedList.m, which is designed to be a browser's delegate. The parts that couldn't be abstracted I put into an object that I chose to call the SortedList's *agent*. So for browsing classes, I created ;TToolsPalette/Utilities.subproj/ClassAgent.m;;¬ClassAgent.m. It turned out that SortedList was an ideal candidate to be palettized itself; so I placed it on the palette (naturally it inherits the List editor that is also part of TTools) as well as a few sample agents. But you don't need to know too much about all of this to try it out - just make sure you've loaded the palette into IB at least once, and then take a look at the ClassBrowsing example. It's worthwhile to note, if you're having trouble with the example, that the file ;Examples/HOWTO;;¬HOWTO outlines in detail how each example was created. The bottom line is, though, that with TTools, one should be able to write virtually any sort of

class browser quickly and easily - in most cases without writing a single line of code!

Q: I've built my palette and it works just fine - in IB. But when I compile an app that uses a nib containing some of my custom objects, the app crashes, or I can't compile it. What's wrong?

A: InterfaceBuilder gets the class definitions of the objects that you assemble in the nib file from the dynamically loaded classes contained in the palette file. This is how it is able to instantiate your objects so that you can use them in "Test Interface" mode. However, when you compile your app, you include the nibs you've created, but the compiled object code is no longer available, since your application doesn't know anything about your palette. Because of this, commercially shipping palettes usually include a library of all the classes used in the palette, along with the header files necessary to work with the objects. You can modify your palette's Makefile.preamble and Makefile.postamble so that a library containing the pertinent classes is generated whenever the project is built. The ;TToolsPalette/Makefile.preamble;;¬Makefile.preamble and ;TToolsPalette/Makefile.postamble;;¬Makefile.postamble files for the TToolsPalette project add a new target, libTTools.a - the OTHER_PRODUCT_DEPENDS symbol now includes libTTools.a, so that the library is made whenever the palette is. The LIBOFILES symbol is

the list of object files that need to be linked into the library; unfortunately, it is impossible to predict what files in a generic project should be included in the library, but particularly the inspector and editor classes should be **excluded**, as well as the "master" palette class as well. Make sure you *include* any custom connector objects, since you'll need them to open any nibs saved with the palette. (Custom connectors are archived along with the other objects in the nib.) It isn't necessary to provide headers for the custom connectors, however; the programmer never needs to talk to a custom connector outside of IB.

Q: What about pre-compiled header files?

In addition to making a library, a shipping palette should also include the necessary header files, and since they are unlikely to change, these headers may be precompiled to save time during the link phase of a compile. The Makefile.preamble and Makefile.postamble files have been further enhanced to cause a master "TTools.h" file to be generated, based on the LIBHFILES symbol, and the resultant header is then pre-compiled into a TTools.p file. The use of the precompiled headers in code is straightforward; simply import the TTools.h header file and the C preprocessor will do the rest. The header information may be conveniently browsed with HeaderViewer, by simply opening the TTools.p file. HeaderViewer will probably complain that it can't find an

introduction for the TTools.p file; to provide HeaderViewer with the documentation it needs for the precompiled headers, use the Info/Preferences menu item, and drag the pop-up list down to the "Documentation Directories" setting. Add the Documentation/TTools directory to the list, and re-open TTools.p, to make all of the documentation available alongside the header files.

Q: I don't understand the documentation for IBEditors, IBSelectionOwners, IBDocuments, etc.; what does an Editor do, and how do I implement one?
Q: How can I implement something similar to the DBModule in IB, where double-clicking on the object seems to bring up another window that allows me to do further configuration of the object, in a way that seems less restrictive and freeform than an inspector? (Ok, it's a loaded question, what did you expect?)

A: IBEditors are the most complex and least-documented of all the IB protocols. Essentially, and IBEditor is an object that allows you to inspect your object in the same way as an inspector, but with more freedom. Because you supply the window and all of the UI for your editor, you can implement pretty much anything you want. The DBModule object in the dbkit palette has an editor that provides a sort of extended connection paradigm, but on a hierarchical basis. The power of IBEditors is even greater when you

consider that the IB protocols provided allow you to implement sub-editors as well. An IBEditor object must respond to a number of IB protocols, most notably the IBEditors protocol. Basically, these protocols define the ways in which IB activates a new editor, closes it, updates it, gets it's internal selected objects, pastes into and out of it, and more. The TTools project contains a small editor for the List class, found in the ;TToolsPalette/ListEditor.subproj/ListEditor.h;;¬ListEditor.h and ;TToolsPalette/ListEditor.subproj/ListEditor.m;;¬ListEditor.m files.

Q: HELP!!! I've been developing my palette for some time, and now I just discovered that I can't open some of the nibs in it! In order to edit the nibs, I need to have my palette loaded. How to I remove the dependencies the nibs have on the palette they are constructing?

A: It's very easy, and rather disturbing, to get into a "circular-reference" game with IB, wherein nibs that are a part of a palette cannot be loaded because they rely on classes that are found in the palette. Basically what's happened is that you've used one of your new palette objects in a nib that is part of the palette project. This is not necessarily bad - these objects are meant to be used, after all. In general, though, you'll probably want to avoid it until you're very comfortable with IB. Once you've introduced such a dependency,

though, it can be tricky to get rid of it. In general, the problem will be that IB seems to have two representations for your class - one which is dynamically loaded with the palette, and one which is parsed in from a header file. If you've made a subclass of Browser (call it TBrowser), then in the nib you're trying to cleanse, a dynamically loaded instance of TBrowser will look like a browser, while an instance that is to rely on the parsed header will look like a custom view (but with a different name, TBrowser). Palette-dependant views, then, are pretty easy to spot. Objects are a little tougher. The best way seems to be to select the object in the objects suitcase, and go to the attributes inspector. If you get your custom inspector for the object, or a "Not Applicable" or "No Inspector", then the object is linked into the app. If you get the "Custom Object" inspector, which allows you to re-class the selected object, then the object is based on parsed (not loaded) code. To remove an object that is of a loaded class and make it of the corresponding parsed, class, make sure that the necessary header is parsed into IB, so that the class appears in the hierarchy; remove the original object (taking note of the connections to and from, if possible); generate a new "Cusom Object" by using the instantiate function in the Classes suitcase to instantiate any parsed class *that is not also loaded* and re-class the object to the original class, using the Custom Object inspector; reconnect as before. If you have no class already loaded, generate one by using the class suitcase to subclass Object; instantiate; copy as above; then switch back to the class suitcase and remove the bogus

subclass of Object that you created by selecting it and hitting the delete key.

Q: My project is getting too large to find things in very easily - what can I do?

A: As projects grow in number of files, the difficulty of finding the header, implementation, interface, image, sound, or other file that you want also grows. One of the dangers of the NEXSTEP environment is that things tend to grow by leaps and bounds - the Timer palette object in TTools, for instance, requires the following files:

Timer.h
Timer.m
TimerInspector.h
TimerInspector.m
TimerIcon.tiff
TimerInspector.nib

And Timer is the most basic object on the palette! The UIBinderList object is implemented with more than 20 different files. Clearly, some subdivision of all this

information is needed. This is where subprojects come in.

The functionality in TTools seemed to divide roughly by palettized object; therefore, I created a subproject for each of Ranker, TBrowser, SwitchView, Timer, and UIBinderList. After that, the correlation breaks down a bit; I palettized several classes from the Utilities.subproj (SortedList, ClassAgent), and provided a new String class and palettized it's agent (StringAgent), both of which I keep in the String.subproj.

How to do it:

Assuming you are starting from an already crowded project, the first thing to do is decide what files belong together. In the case of TTools, this was easy; an object, it's inspectors, editors, connection inspectors, and supporting code for these files belonged in the same subproject. More generally, a given subproject will often contain a single nib (or a number of closely associated nibs) and all of the supporting code that manages that nib.