TTools (2.0)

by Thomas Burkholder, NeXT Developer Support Team

Valid for 3.1 and later.
Not valid for 3.0.

Goals

1. To provide an educational example of advanced palette-building techniques.
2. To address issues associated with managing a large palette project.
3. To push the envelope of IB tools.

Overview

This MiniExample is an advanced IB Palette primer. The prerequisite knowledge for this is a good understanding of Interface Builder and Objective C including knowledge of the Protocol and Category

language elements. The palette illustrates:

· How to palettize a simple view, window, and object.
· How to provide your palettized view or object with an inspector in IB.
· How to write a custom connection inspector and custom connector objects for your palettized object.
· How to access your superclass's inspector.
· How to effectively palettize Matrix without restricting its prototype cell class.
· How to palettize an object that uses the target/action connection paradigm in IB.
· How to browse the class hierarchy.
· How to implement a complete object editor.
· How to generate a library from your palette project so that applications can link in the palette code.
· How to automatically generate a library .h file, and precompile it.
· How to bind to instance variables that are not of type id (hyper advanced).

Topics Of Interest

How To Get Started::

First, drag the entire TTools directory into some known location within your home directory. Open the PB.project, change the target to "install", and click build. Note that this will create a ~/Library/Palettes, if you don't already have one; in that it will put the palette, a Documentation/TTools directory, the libTTools.a library, and a Headers/TTools directory. Once you've done this, you can change the target to "clean", and do a build to regain the space used by the object files during the build. When you're ready to use the palette, just double-click on it in your ~/Library/Palettes directory. For a quick look at what the palette can do, take a look at the files in the Examples directory. Any questions? Take a look at the HOWTO document in the Examples directory, which outlines how each example was built. Try reading the relevant parts of this document, the QA.rtf document, and any relevant parts of the class documentation you'll find either in the project directory under Documentation/TTools or in your ~/Library/Palettes/Documentation/TTools directories. If you're still stuck, drop me a line at thomas@next.com and I'll see if I can make it a bit clearer. A lot of this stuff is really advanced; if you're a relative neophyte, your best route to figuring it all out will be to try to learn about the objects in

the order that they're presented in this document. And unless you think you really understand IB well, don't try to dissect TBinderList at all.

Timer:

Timer is intended to be a non-trivial palettized object example. An instance of the Timer class calls an action in it's target at specified intervals. In addition, the Timer instance stores a value, so that it can keep track of some state. Hooking up the -takeIntValuFrom: method of a TextField object to the action of a Timer instance will allow the TextField to display the timer's current value, for example. In addition, because the timer's value can increment by an amount independant of it's period, and has the ability to "wrap" at a certain value, it can be used as a "clock" in many situations. Timer's action methods are fairly simple - one can tell a timer to start:, stop:, pause:, and resume:. It is possible, for instance, to connect three Timer instances to three TextFields, and have a fairly accurate real-time clock.

WhizzyInfo:

This is a short example showing how to use two of the object provided, the Timer object and the SwitchView object, for simple animation purposes, such as is often found in application info panels. To observe it's functionality, simply open it's main nib, and use the test-interface mode. The example is included as a complete project as a demonstration of how to link in the library created by the palette project; the functionality of the example is entirely demonstrable within IB.

In this example a Timer object is hooked up to a SwitchView object that is connected to several buttons, each of which shows a person's picture. The Timer instance is configured to start it's value at 0, increment by 1, and wrap at the value 5. The timer's target is connected up to the SwitchView instance, and the action selected is "takeViewNumberFrom". The SwitchView instance is configured to use the "-intValue" callback, so that the views should be shown in order in the content area of the SwitchView. Finally, two buttons are needed to start and stop the Timer instance.

Ranker*:*

Ranker is a matrix subclass whose added functionality is similar to that of "NiftyMatrix" (from the ScrollDoodScroll MiniExample.) The main teaching purpose of the palettized Ranker class, however, is to solve the problem of subclassing Matrix in IB. The problem is that there is no way to tell IB that you want your subclass of Matrix to be used when an alt-drag is done on a control. IB will always use the vanilla Matrix class. Often, palette builders will provide a palette containing many instances of their matrix each populated with a different cell class - a matrix of SliderCells, a matrix of TextFieldCells, a matrix of ButtonCells, and so forth. This not only takes up a lot of room on a palette, but it doesn't allow new cells (provided in other palettes) to be used with the new Matrix subclass.

To solve this, Ranker's IB code asks the user for a Cell class when the Ranker is first dropped onto a window. It does this by displaying a class hierarchy (rooted at the ActionCell class) in a panel, and allowing the user to select the desired prototype cell. After the selection has been made, Ranker will assign it's prototype to be an instance of the selected class, and will instantiate itself upon the window that it was dragged onto. Because the Ranker asks for it's prototype at drag-time, there need only be one instance of Ranker on the palette, and because the class-hierarchy-browser that the user selects cell classes from is dynamic, it is able to use cell classes that are dynamically loaded from other palettes.

SwitchView:

This is probably the most challenging and interesting of all the palettized objects provided. The SwitchView class is intended to allow simple view-swapping to occur, such as one often wants to do when implementing an inspector paradigm. The logical process to follow in IB in such a case is to connect all the views that you want to show to outlets of the same managing view, which will display a different view according to some stimulus. For example, InterfaceBuilder's inspector panel operates in such a fashion - the PopUpList at the top of the inspector is the control that causes several different views to appear in the box beneath it.

The major problem with implementing this is that your managing view only has a certain number of outlets to connect to - really what you want is for your managing view to keep a list of views that can be variable in number. However, IB contains no connection paradigm to allow this. Enter the custom Connection inspector, something the writers of IB thoughtfully provided to allow the palette builder to customize the way in which custom objects are connected to eachother. In this case, we implement a connection inspector that allows the user to connect any number of views to our SwitchView instance.

This is fairly simple programmatically, but is not entirely obvious from the documentation.

The method by which the connection inspector is supplied is very similar to the way that standard attributes inspectors are supplied - via a "getConnectInspectorClassName" method and associated module. The connection inspector makes heavy use of various calls to the IBDocument-protocol-conforming object acquired through the [NXApp activeDocument] call, and also manages a number of IBConnector-protocol-conforming objects (SVConnector instances).

Another area of interest in the SwitchView class is it's callback ability. The control which swaps views in and out of it's content area needs to be connected to the "takeViewNumberFrom:" method in SwitchView. When this method is called in a SwitchView instance, a callback to the sender is done. If the SwitchView instance has been left in "automatic" mode, it will figure out what selector it should use to call in it's sender in order to determine what view to swap in. If the sender is a matrix with more rows than columns, for instance, the SwitchView instance will call "-selectedRow". If the sender is a slider, it will call "-intValue". As a last resort, if the sender does not fit into any other category, the SwitchView instance will use the "-tag" method to call back for the value. If the SwitchView instance is in "manual" mode, then the method name to be called will have been set in IB, via the SwitchView's

attribute inspector. This makes SwitchView very general in nature, such that any object that has a no-arguments method that returns an integer can be used as the control for the SwitchView.

Because SwitchView is a subclass of Box, it has all of the Box functionality. However, it has it's own inspector, which over-rides the Box inspector. Rather than re-implement the Box inspector, however, you can, through a neat, four-line trick in the -getInspectorClassName method, have your cake and eat it to:

```
- (const char *)getInspectorClassName
{
        NXEvent *ev = [NXApp currentEvent];

        if (ev->flags & NX_ALTERNATEMASK)
                return [super getInspectorClassName];
        else
                return "SwitchViewInspector";
}
```

The above is the method in it's entirety. It allows you to use the superclass inspector by alt-clicking the object you want to inspect. This is a big time-saver!

List:

Basic information about the List class can be found in the on-line documentation. The List object available in the TToolsPalette is in fact the exact same object. All that's been added is the ability to set up a list in IB to contain objects of your choice. To use the List object, drag the instance from the palette into your objects window; double-click on the newly instantiated object to get it's inspector. Select the class of object you'd like to add to the list in the top browser, and then click the add button to add a new instance of that object. Notice that the inspector reflects what you have selected in your editor - you can add a new Timer instance, for example, and be able to inspect it as well. You'll also be able to cut, copy, and paste objects within the editor and between the editor and the rest of Interface Builder as well. Notice that you can create a new list as an element of your list, double click on it, and get a sub-editor for it, providing a potentially infinite level of detail for lists of stored objects. IBEditor functionality is extremely powerful, but requires a fairly thorough knowledge of IB's inner workings;

hopefully the ListEditor example will be useful.

SortedList*:*

This is a subclass of List that inserts elements in sorted order. To do this, the addObject: method is over-ridden, so that a binary insertion may be done. Since the method of comparing an element in the list and the candidate element is entirely dependant on what objects are being stored in the list, the sorted list asks the object connected to the *agent* outlet, using the compare:with:sender: method in the SortedListAgent protocol. These "agent" objects are meant to have object-specific information about how to compare and display particular objects, among other things. Several sample agents are provided, including a ClassAgent (used for class browsers) and a String Agent (used for a custom string object, also included). In addition to providing sorted insertion, SortedList is meant to act as a browser's delegate; in this way, using the List editor, one can set up a SortedList in IB, add the desired objects, change each object with it's inspector, attach the list to the appropriate agent (StringAgent, for instance, is asked "displayStringFor:anObject sender:sender", for which it does a : "return [anObject stringValue]") and using Test Interface mode, immediately view the resulting browser/list combination.

This tends to eliminate a lot of code.

In addition to providing this functionality on the palette itself, all of the various inspectors in TTools that use browsers to display information use the SortedList or SortedStorage classes - examples of how to use the Browser/SortedList/Agent combinations are the palette inspectors and editors themselves! To understand how to correctly use the SortedList, StringAgent, and ClassAgent objects, see the ListBrowsing example.

TBinderList:

Or "Toward graphical programming". Users of TTools should be warned up front; TBinderList is extremely experimental and potentially unsafe to use. Moreover, it contains a high level of abstraction, and is currently under-documented. It is an attempt at binding an object's instance variables of any type to user-interface objects. What does this mean to the IB user? The net result is that the user will be able to make, for instance, a TextField, display the contents of an instance variable in a data-bearing object. The converse is also true; often, we'll want the user to be able to enter a string in a TextField, hit

enter, and have an instance variable (probably a char *) in some object to have changed as a result. All of this is accomplished through the use of the TBinderList, TBinder, and lots of IB-inspection code that allows it all to be done in IB and saved into a nib. Users of TTools should note that it is meant as an advanced educational tool, and not as a product. There are probably some bugs (no IB-crashers, I *think*...), and some rough edges. I'll try to include an example of how to do this, but I expect documentation to be minimal for this release. A thorough understanding of the rest of the palette code is necessary before attempting to understand what's happening in this part of the palette.

Other Classes and Categories:

A category of object, called ClassAdditions, is used to provide information about classes and method selectors not readily available. For those interested in burrowing into the maze of the run-time, this category may be very useful. For dedicated run-time fiends, the TBinder and SelPairAgent objects should provide some new tricks.

Makefile stuff:

The TTools product automatically generates it's own libTTools.a library, TTools.h header file, and TTools.p precompiled header file, through the use of some Makefile hacking. The principle location of the added rules is in the main Makefile.preamble and Makefile.postamble files.

Changes since TTools 1.0:

Basically everything. The Ranker and Timer objects are the same, the AlertPanel object I threw away, and everything else is new. Timer changed a bit; versioning was used, so old instances of Timer should be fine. Also, this version works only in 3.1, not in 3.0. It requires minimal Makefile hackery to make it work under 3.0, but I decided the project was large and confusing enough as it is.

Kudos:

Special thanks to Mike Gobbi and others for coming up with the alt-click inspector trick, and to all our customers who asked me the interesting questions that resulted in this MiniExample. Also, thanks to

Alan Ethanson who helped me with the TTools.h generation sh-code.

Disclaimer:
You may freely copy, distribute, and reuse the code in this example. NeXT disclaims any warranty of any kind, expressed or implied, as to its fitness for any particular use.