Stone Age n
(1864) *:the first known period of prehistoric human culture characterized by the use of stone tools*

Welcome to the Stone Age!   This column shall be dedicated to a recursive descent into appkit programming: stone tools *aka* Acme Gizmos, new objects, tricks, and tales from the dark night of the programmer's soul. In the spirit of FSF and my conviction that this really is a revolution, here's a tool from my chest. Like all good object libraries, this column should be written collectively. I invite all of you to "Step right up" in this space and show off your neat objects, even if unpolished, because its the concepts that count.   Next month: Kris Jensen describes a new Gizmo.

Composite Objects: A Slider/TextField Hybrid

SliderDualActing and its cell class, SliderCellFine, combine a special ªdual-actingº slider, a text field, and two arrow buttons into one composite object.   This allows the end-user multiple input techniques while abstracting the client or application user of the object away from the details of coordinating the various Appkit components. The special slider solves some of the problems of providing fine control over a large range of values in a small space. This project illustrates the power gained both by subclassing Appkit objects and combining objects.

First, a picture:

Features

Component objects make programming easier:

· The end-user gets multiple input methods: a slider, a textfield, and a pair of buttons which decrement or increment the slider, while the program only deals with one object. All of the validation and synchronization methods are in one place instead of distributed throughout source files. Programs often have lines like these in read: and other resetting methods:

```
[numberField setIntValue:number];
if (number>[numberSlider maxValue]&&okToExceedMaximum)
      [numberSlider  setMaxValue:number];
[numberSlider setIntValue:number];
```

All of the coordinating code can be replaced by the last line.   The SliderDualActing handles the rest.

·€We want to continually update the textfield as we drag the slider (and SliderDualActing handles this without any programmer intervention), but we also want to notify another target either continually or on mouseUp. The second case is more usual, as continual notification may take too much processing time. Therefore, our composite object has an *UpTarget* and an *UpAction*.

· Users should not be stopped from entering values higher than *maxValue* or lower than *minValue*, if it makes sense to the application. For example, in TextArt, the fontSizeSlider has a default value of 140 points. However, if users want a higher value, they simply type in a value into the textPal, and if the SliderDualActing BOOL instance variable *allowHigher* is YES, the slider resets its maximum value. A further enhancement to this class would be methods to write the user's maximum and minimum values to the NX_Defaults mechanism to restore on the next launch.

· Sometimes, the user justs wants to ªcrawlº along the slider's values. I have added a matrix of two buttons which increment or decrement the slider by *altStep*. They connect to the sliders' action method **incrementDecrement:**.

· Component objects can provide a convenient method for implementing **Undo**. The SliderDualActing knows its last value and can respond to an undo method. Alternatively, you can specify an undo target and a tag telling the target what needs to be undone. The default is for the slider to handle undo itself. One simple strategy for a single-level undo is to have a global undo object, which stores the id of the last control set. Undo from the menu tells this undo object to send the undo method to the control that last set it. My undo object is accessable via NXApp, and looks like this:

```
@interface Undo:Object
{
    BOOL  hasUndo;
    id  lastClass;
}
- setLastClass:anID;
- undo:sender;
@end
```

Special sliders provide several additional user interface features:

· Sliders with a large range can cause problems: If the slider isn't very long, the resolution (how much the value changes when the slider is dragged one pixel) can become unacceptably large. On the other hand, screen space is often in short supply, making it hard to use the long sliders needed   to obtain a fine resolution. SliderDualActing and SliderCellFine provide methods to change the slider's value by specifiable small amounts by checking if the Alternate or other meta keys are down while the slider is being dragged. Currently, pressing the Alternate key causes the slider to change by the *altStep* instance variable and adding the Shift key halves this amount.

· Looking at float values that have insignificant digits is ªnoisyº and very unSteveLike™. Sometimes, however, a user wants to specify a value to a high degree of accuracy. Our composite object provides a mechanism to dynamically change the TextField **textPal**'s floating point format.   Normally the user will not be bothered with the floating point values, but dragging the slider while pressing the Alternate and Shift keys puts the SliderDualActing into decimal state (as well as changing the value by a prespecified small amount). This allows precision while preserving aesthetics.

· The user can reset the slider to its various defaults (maximum, minimum, number of decimal places displayed) while using the program by clicking the slider while pressing the Command key.

Using SliderDualActing, a step by step guide in Interface Builder:

The first time the object is used, perform steps 1 thru 13; thereafter only step 13 is necessary:

0] Assume that you have a class with an instance variable named ªsliderDAº
1] Copy SliderDualActing.[hm] and SliderCellFine.[hm] to your project directory.
2] Bring up the Class Window (Command-5) and the Class Attributes Inspector (Command-1)
3] Make a subclass of slider:
    a] Traverse the Class hierarchy to Object->Responder->View->Control->Slider
    b] Select SubClass from the Operations pull-down menu in the Class Window
    c] Rename the subclass to SliderDualActing
    d] Select Parse from the Operations pull-down menu and answer ªOKº to the ªAdd SliderDualActing to Projectº dialogue box
4] Create a new custom view by dragging one from the Palette Window
5] Make it a SliderDualActing by bringing up Inspector Attributes Window (Command-1) and clicking on the SliderDualActing class name
6] Size the new slider: Bring up Inspector Size Window (Command-4) and type in a width of 16. if it's to be a vertical slider, or a height of 16 if it's a horizontal slider.
7] Drag a textField from the Palette Window
8] Drag a button from the Palette Window. Make a matrix of two buttons by dragging while pressing the Alt key. In the Inspector Attributes window, be sure to check ªCells Tag = Positionº. Add arrow icons to these buttons.
9] Select the three objects and ªboxº them by typing Command-g. This allows you to copy and
paste the control while retaining the connections among the three objects.
10] Connect the textPal outlet of sliderDualActing to the textfield.
11] Connect the action of the textfield to the SliderDualActing method: ª*sendTextAction*:º
12] Connect the matrix' target to the SliderDualActing method ª*incrementDecrement*:º

13] Connect your control object's sliderDA outlet to the sliderDualActing view. Now, for other instances of SliderDualActing, you can copy and paste this box, and only need to set your application's control object to the slider itself.

In your code, in the ªappDidInitº method sent to the App add some initialization code:

You need to set the slider's UpTarget and UpAction and its various default values. An excellent place for this is after the Application receieves an **AppDidInit**: message. For example:

```
/* The app has an outlet  named "sliderDA" in this example */
- appDidInit:sender
 {
   [[[[sliderDA setUpTarget:self action:@selector(setAngle:)]
             setMax:360. allowHigher:NO min:-360. allowLower:NO]
             setAltStep:1. whole:YES default:0]
             setFormat:NO left:1 right:3];           // formats text pal
   /* other initialization code here */
   return self;
}
```

Caveats:

Problem: IB has no inspector for subclasses of known objects in V1.0. This means you lose the ability to specify various defaults in IB.
Fix: Call the initialization routines illustrated above and documented in SliderDualActing.m.

Problem: If the value of altStep is large compared to the range of the slider, drawing update anomalies occur. Instead of adding resolution, you subtract it. SliderDualActing should have more error checking for bad parameters.

The archiving methods have not been tested since I read mine in from a nib file.

Documentation and Files:

The documentation includes Class Specifications for SliderDualActing and SliderCellFine. I really enjoyed "cloning" the superclasses in WriteNow and just cutting and pasting my methods into the preformatted pages. It looks ªjust like the book.º I have a deep appreciation now for the great insight and hard work that went into producing the Appkit and its extraordinary documentation, definitely a winning aspect of the NeXT programming environment.

· SliderDualActingDistribution (available in NeXT archives everywhere):
SliderDualActing.doc.wn
SliderDualActing.h
SliderDualActing.m
SliderCellFine.h
SliderCellFine.m
SliderCellFine.doc.wn

SliderDualActing.nib
SliderDualActingDemo.m
SliderDualActingDemo.h
Assorted:README,Makefile,IB.proj,snds,tiffs.