It Just Kicks In

Doug Hennig

The best kind of tool is one that automatically does its job without having to set any properties, write any code, or even lift a finger. This column looks at a strategy for implementing these types of tools, and provides a popup calendar for Date fields and a universal find function for your applications.

One of the things I love about object orientation is inheritance. If you make a few tweaks to a class, any subclass or instance of that class automatically gets the new behavior or appearance without touching them. I refer to this phenomenon as "it just kicks in" because the new behavior just kicks in without any effort (other than making the change to the class in the first place). This was really driven home recently when I added a powerful new feature to our base classes (the "find" functionality described in this column) and this new feature "just kicked in" on every form a co-worker had created for a client application the next time she rebuilt the project. She was absolutely stunned at how little effort there was on her part (none, actually <g>).

My idea of the ideal tool is one that "just kicks in" like this. We won't often hit this ideal, because some tools are just too complex or unwieldy to add to our base classes. However, in this month's column, we'll look at a strategy for implementing "just kicks in" tools, and then look at a couple of extremely useful tools of this genre: a pop-up calendar for Date or DateTime fields and a universal "find" feature for your applications.

The Strategy

Before we start, I'd like to publicly acknowledge the debt of gratitude I have to two FoxPro gurus: Steven Black and Ken Levy. The ideas expressed in this column were either shamelessly stolen from them or were inspired by their ideas. Thanks especially to Steven for introducing and making more understandable some of the design patterns used in these tools.

The strategy I took to create new functionality in our TextBox base class (SFTextBox, described in my February 1997 column, and included in the Subscriber Downloads in SFCTRLS.VCX) is based on a shortcut menu. Shortcut menus, which are typically displayed when the user right-clicks on an object, are really easy to do in VFP 5 thanks to the new SHORTCUT clause in the DEFINE POPUP command. Here's an outline of how this strategy works:

- The RightClick() method of SFTextBox calls a custom ShowMenu() method.

- ShowMenu() defines a shortcut menu called SHORTCUT (rather than using the Menu Designer to create a shortcut menu file, I added the code directly to SFTextBox so it's as self-contained as possible), and then calls the custom method ShortcutMenu() to populate the menu. Why not put the code to populate the menu directly into ShowMenu()? So you can subclass SFTextBox or even change the behavior of a single instance to create specialized versions that override the contents of the shortcut menu. The problem with putting all the code into ShowMenu() is that you can't add new items to the shortcut menu without copying and pasting code from the base class, since the menu population code must be preceded by menu definition code and followed by menu execution code. Putting this code into separate methods makes creating specialized versions easy.

- SFTextBox has a custom property called oHook, which provides the capability for hooking objects together. While a complete discussion of hooking objects is beyond the scope of this article, here's a brief description. oHook normally contains .NULL.; however, you can store a reference to another object in oHook (this might be done in the Init() method of an instance or subclass of SFTextBox, for example, or perhaps by another object such as a form the instance sits on). After calling ShortcutMenu(), ShowMenu() checks to see if oHook points to a valid object and if that object has a ShortcutMenu() method. If so, that object's method is called. This provides the ability to hook an instance or subclass of SFTextBox to another object that adds additional items to the

shortcut menu. We won't use oHook in this article, but we might use it in future articles to add additional features to objects without having to subclass them or modify their base behavior.

- After the menu has been populated with ShortcutMenu() (and possibly a hooked object's ShortcutMenu()), ShowMenu() activates the menu.

Here's the code for ShowMenu():

```
local loObject

* Define a reference to this object in case the action
* for a bar is to call a method of this object, which
* can't be done using "This.Method" ("This" isn't
* applicable in a menu). The action can be specified
* as "loObject.Method".

loObject = This

* Define the menu.

release popup SHORTCUT
define popup SHORTCUT shortcut from mrow(), mcol()
with This

* Populate it using a custom method.

  .ShortcutMenu()

* Use the hook object (if there is one) to do any
* further population of the menu.

  if type('.oHook') = 'O' and not isnull(.oHook) and ;
    pemstatus(.oHook, 'ShortcutMenu', 5)
    .oHook.ShortcutMenu()
  endif type('.oHook') = 'O' ...
endwith

* Activate the menu if necessary.

if cntbar('SHORTCUT') > 0
  activate popup SHORTCUT
endif cntbar('SHORTCUT') > 0
release popup SHORTCUT
```

The ShortcutMenu() method can add anything you wish to the SHORTCUT popup. Here's what I have in mine:

- Cut, Copy, Paste, Clear, and Select All: these functions are easily implemented using the new SYS(1500) function (assuming you have an Edit pad in your menu bar).

- Calendar: if the textbox holds a Date or DateTime value, a Calendar bar appears. Choosing this bar calls the custom ShowCalendar() method of SFTextBox. Note that ShowCalendar() and other methods of SFTextBox have to be called using a variable (loObject, which is defined in ShowMenu()) rather than "This" as the object name, because "This" isn't valid in menu code, which executes outside the scope of the object.

- Find: SFTextBox has a custom property called lFind that's set to .T. by default. If lFind is .T. and the object's ControlSource is filled in, ShortcutMenu() adds a Find bar to the menu. This bar calls the custom FindField() method when it's chosen.

Here's the code for ShortcutMenu():

```
local lcRow, ;
  lcCol, ;
  lnBars
with This

* Calculate current row and column coordinates as
* strings so we can macro expand them.

  lcRow = ltrim(str(Thisform.Top  + .Top  + mrow()))
  lcCol = ltrim(str(Thisform.Left + .Left + mcol()))

* Add cut, copy, paste, clear, and select all items to
* the shortcut menu.

  define bar 1 of SHORTCUT prompt 'Cu\<t'
  define bar 2 of SHORTCUT prompt '\<Copy'
  define bar 3 of SHORTCUT prompt '\<Paste'
  define bar 4 of SHORTCUT prompt 'Cle\<ar'
  define bar 5 of SHORTCUT prompt '\-'
  define bar 6 of SHORTCUT prompt 'Se\<lect All'
  on selection bar 1 of SHORTCUT ;
    sys(1500, '_MED_CUT',   '_MEDIT')
  on selection bar 2 of SHORTCUT ;
    sys(1500, '_MED_COPY',  '_MEDIT')
  on selection bar 3 of SHORTCUT ;
    sys(1500, '_MED_PASTE', '_MEDIT')
  on selection bar 4 of SHORTCUT ;
    sys(1500, '_MED_CLEAR', '_MEDIT')
  on selection bar 6 of SHORTCUT ;
    sys(1500, '_MED_SLCTA', '_MEDIT')

* If this field contains a Date or DateTime value,
* add a Calendar bar.

  lnBars = 6
  if type('.Value') $ 'DT'
    define bar 7 of SHORTCUT prompt '\-'
    define bar 8 of SHORTCUT prompt 'Cal\<endar'
    on selection bar 8 of SHORTCUT ;
      loObject.ShowCalendar(&lcRow., &lcCol.)
    lnBars = 8
  endif type('.Value') $ 'DT'

* If we can find on this field and we have a
* ControlSource, add a Find bar.

  if .lFind and not empty(.ControlSource)
    define bar lnBars + 1 of SHORTCUT prompt '\-'
    define bar lnBars + 2 of SHORTCUT prompt 'Find...'
    on selection bar lnBars + 2 of SHORTCUT ;
      loObject.FindField()
  endif .lFind ...
endwith
```

That's it for the strategy part. You can easily add additional menu items if you wish. Adding them to SFTextBox.ShortcutMenu() means they'll be available to all text boxes. You can also subclass SFTextBox and have ShortcutMenu() add additional items to the menu using code like:

```
dodefault()
lnBars = cntbar('SHORTCUT')
define bar lnBars + 1 of SHORTCUT prompt 'Whatever'
on selection bar lnBars + 1 of SHORTCUT ;
  loObject.MyCustomMethod()
```

You can also extend the behavior of SFTextBox by storing a reference to an object with a ShortcutMenu() method into the oHook property; that object's ShortcutMenu() could add anything to the SHORTCUT popup.

Implementing the Calendar

ShortcutMenu() adds a Calendar bar to the shortcut menu if the textbox contains a Date or DateTime value. Choosing this bar calls the custom ShowCalendar() method of SFTextBox. Here's the code for that method:

```
lparameters tnRow, ;
  tnCol
local loCalendar
loCalendar = NewObj(This.cCalendarClass, This)
with loCalendar
  .Value = iif(empty(This.Value), date(), This.Value)
  .Top   = tnRow
  .Left  = tnCol
  .Show()
endwith
```

You're probably thinking "Wow, is that it?" Not exactly. There are several things happening here:

- SFTextBox has a custom property called cCalendarClass. This property contains the name of a class to instantiate for our calendar. By default, cCalendarClass contains "SFForms,SFCalendar" (we'll talk about this in a moment). The advantage of storing the name of the class in a property rather than hard-coding it into the method is that you can change the class to use in the Property Sheet for a subclass or a specific instance of SFTextBox, or even change the value at run time (say certain users get one type of calendar and other get a different type, or perhaps DateTime fields get a special dialog that includes an analog clock).

- Notice there's no CREATEOBJECT() command. Instead, I use a tiny program called NEWOBJ.PRG (included in the Subscriber Downloads) to create the object and return a reference to it. NEWOBJ takes a parameter containing the name of the class to instantiate, optionally preceded by the name of the VCX the class is in and a comma. NEWOBJ can also accept additional parameters; these are simply passed to the Init() method of the object being created. The advantage of including the VCX name with the class name is that you don't have to worry about whether you've done a SET CLASSLIB to the particular VCX or not; NEWOBJ will take care of that detail for you.

- Because cCalendarClass contains "SFForms,SFCalendar", we obviously must have a class library called SFFORMS.VCX which contains a class called SFCalendar. This VCX is included in the Subscriber Downloads. I won't bother going over the definition of this class other than to say that it's a modal dialog containing a Calendar ActiveX control (one of the ActiveX controls that comes with VFP 5) and OK and Cancel buttons

- After the SFCalendar object is created, its Value is set to the date contained in the SFTextBox (or the current date if nothing was entered yet) and its Top and Left properties are adjusted so the dialog appears near the textbox that displayed it. The calendar is then displayed using its Show() method. SFCalendar has code that puts the date chosen by the user in the calendar back into the Value of the SFTextBox when the user double-clicks on a date or chooses OK.

Implementing the Find Functionality

ShortcutMenu() adds a Find bar to the shortcut menu if the textbox has a ControlSource and its custom lFind property is set to .T. (which it is by default, meaning any SFTextBox bound to a field in a table or view will automatically have find functionality). Choosing this bar calls the custom FindField() method of SFTextBox. Here's the code for that method:

```
local loFind
with This
  loFind = NewObj(.cFindFieldClass, .ControlSource, ;
    Thisform)
  loFind.Show(1)
```

```
endwith
```

As with ShowCalendar(), FindField() is deceptively simple. It uses NEWOBJ.PRG to create an object of the class specified in the custom property cFindFieldClass (which contains "SFForm,SFFindFieldForm" by default), passing to the new object the ControlSource for the textbox (so it knows what field to search on) and a reference to the form the textbox sits in (so it can refresh the form when a record is found). As with SFCalendar, I won't show the code for SFFindFieldForm (you can check it out yourself, since it's part of the Subscriber Downloads), but will briefly describe it:

- SFFindFieldForm consists of three objects: Find Next and Cancel buttons and a container object called SFFindField (defined in SFCCTRLS.VCX). SFFindFieldForm accepts two parameters: the aliased name of the field to search in and a reference to the form to refresh when we find a matching record. These two parameters are simply stuffed into custom properties of the SFFindField object, which has almost all the functionality of SFFindFieldForm.

- SFFindField is a container object with two controls: a textbox where the user can enter the value to search for, and an option group where the user can indicate whether the value should be found at the start of a field or anywhere within the field (as you can probably guess, we're going to do something like LOCATE FOR &lcField = lcValue or LOCATE FOR lcValue $ &lcField, depending on how the option group is set).

- SFFindField has a custom FindNext() method which is called by the Find Next button in SFFindFieldForm. FindNext() searches in the specified field for the value the user entered, and refreshes the form if a match is found. It stores whether it found a match or not in a custom lFound property. If lFound is .T. (meaning a match was previously found and the user wants to find the next one), FindNext() uses CONTINUE. If not, it uses a LOCATE FOR as described in the previous point. Thus, it acts likes the Find Next button in Find dialogs such as those in VFP and Word; pressing Find Next finds the first match, and continuing to press Find Next cycles through all matching records.

Why did I create an SFFindField container only to drop it on a form? Why not just put the controls directly onto SFFindFieldForm? The reason is that I can envision other users for SFFindField. For example, I might want to have a Find function for the form. Perhaps it brings up a dialog containing a combo box of fields the user can search on and a SFFindField object in which the user can specify the value and how to search. Since the searching capability is encapsulated within SFFindField, all this new dialog needs to know is that it puts the name of the field chosen in the combo box into the cField property of the SFFindField object and calls the FindNext() method when the user clicks on a Find Next button.

You could improve upon SFFindField by supporting SEEK rather than LOCATE FOR. I chose not to since this would complicate matters; you'd likely have to add properties to SFTextBox such as cSeekExpression (since the index expression for a field may not be just the field name, but something like UPPER(<field>)) and cTag (the name of the tag to search on), and set them for each field on a form. Remember, one of my design decisions was that it "just kicks in" without having to set any custom properties of any object on any form.

Every user I've demonstrated this feature to thought it was very cool, because the user interface is very simple: right-click on a field, choose Find, enter a value, and click on Find Next to bring up records containing that value in that field one at a time. Developers like it even more because with no effort (once the changes are made to SFTextBox and the other classes are created), you can add this feature to all your existing forms. It just kicks in!

Example

The TESTFORM form in the Subscriber Downloads is a very simple form; it just has a few fields from a PRODUCTS table. However, right-click on any field and check out the Cut, Copy, Paste, Clear, and Select All functionality. Choose Find and see how the find feature works. Right-click on the Last Order date field and choose Calendar to see how the calendar works.

To create the form, I simply set SFTextBox as the control to create for certain data types in the Field Mapping page of the Tools Options dialog, dragged some fields from the DataEnvironment to the form, and created simple Previous and Next buttons. No custom properties to set, no custom code to write. It just … OK, I think you get the point.

Summary

The changes we made to SFTextBox are summarized in Table 1. Of course, you don't have to use SFTextBox; you could apply these same changes to your own TextBox base class.

Table 1. Changes to SFTextBox.

Method/Property	Change or Purpose
RightClick()	calls This.ShowMenu()
ShowMenu()	custom method to define, populate, and execute a shortcut menu
ShortcutMenu()	custom method to populate the shortcut menu
ShowCalendar()	custom method to instantiate a calendar class
cCalendarClass	custom property containing the name of the calendar class
FindField()	custom method to instantiate a find class
cFindFieldClass	custom property containing the name of the find class
lFind	custom property; .T. if the user can find on this field
oHook	custom property containing a reference to a hooked object that extends the shortcut menu

I hope this article inspires you to create your own tools, and that you find the calendar and find features as powerful yet easy to implement and use as we do.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, and will be speaking at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). 75156.2326@compuserve.com or dhennig@stonefield.com.