Objectify Your Menus

Doug Hennig

Object-oriented menus have been one of the most commonly requested enhancements to VFP. This month, Doug presents a set of classes that allow you to implement object-oriented menus right now.

Although VFP is a great object-oriented development language, there are two glaring areas of the product that have never been objectified: reports and menus. In my June 1999 column ("Report Objects"), I presented a set of classes that allow you to create reports using objects. This month, we'll do the same thing for menus.

What's the big deal about object-oriented menus? After all, it'd be extremely rare for a menu to be reusable from one application to another. However, what about parts of a menu? Individual bars or even entire pads could definitely be used in many applications. For example, the bars in the Edit pad (Undo, Redo, Cut, Copy, Paste, Clear, and Select All) are usually the same from application to application. Some bars in the Help pad (such as Help and About) and File pad (such as Print Setup and Exit) would likely call the same functions in every application. So, being able to reuse parts of a menu would be a time saver for many developers.

Another benefit of object-oriented menus is inheritance. You might have several bars that are similar (for example, they may have the same SKIP FOR clause or even call the same function, passing a parameter to indicate which one was chosen) but have different prompts. Wouldn't it be nice to create a new bar by subclassing an existing one and just changing the few things that make it unique?

How Menus Work in VFP

Before we look at the classes that implement an object-oriented menu, let's review how menus are handled in VFP. A menu bar appears below the title bar of an application. Although you can define a menu bar using DEFINE MENU, most developers use the VFP system menu bar, _MSYSMENU, because it's easier and that's what code generated by the Menu Designer does.

A menu bar consists of pads. A typical application has File, Edit, and Help pads at a minimum, but Tools, View, and Window pads are also common. Pads are created with the DEFINE PAD command. A pad has a prompt, a hot key (usually an Alt-key combination), a SKIP FOR clause (which, when it evaluates to .T., disables the pad), and a message that appears in the status bar when the pad is highlighted. Pads can have other properties, such as font and color, but these are rarely used. You can specify what should happen when a pad is selected, but the standard behavior is to display a popup.

As with pads, popups can have a number of properties specified with the DEFINE POPUP command, but the two that are typically used are MARGIN, indicating that extra space is available at the left margin of the popup so images or mark characters can be displayed, and RELATIVE, which means that items in the popup appear in the order in which they're defined.

Popups consist of bars, the items a user can select from the popup. A bar is created with the DEFINE BAR command, and has a number of properties you can specify, including the prompt, hot key and the text representing the hot key that should appear to the right of the prompt, a SKIP FOR clause, and a message that appears in the status bar when the bar is highlighted. As with pads, other bar properties such as font and color are rarely used. VFP 7 adds several new bar properties: the name of a file containing the image to display to the left of the bar's prompt, the name of a system menu bar whose image should be used for the bar, whether the bar should appear inverted, and whether the bar should be an MRU (most recently used) bar (it appears as a chevron pointing downward). These new properties allow you to create Office 2000-style menus in VFP. You specify what should happen when the user selects a bar with the ON SELECTION BAR command.

Now let's look at the classes that implement object-oriented menus. As I mentioned earlier, some of the options VFP provides for menus are rarely if ever used. I decided when I created these classes that I'd ignore options I never use or those that don't follow Window standards (many of these options are carry-overs from the DOS days). I also decided for simplicity to ignore cascading menus for now (a cascading menu has bars that, when selected, display a subpopup of choices). All the classes discussed in this article are located in SFMENU.VCX.

I'd like to give credit to my sources of inspiration (and a bit of code <g>) for these classes: the Visual FoxPro 3 Codebook by Yair Alan Griver (Sybex, ISBN 0-7821-1648-5) and Visual FoxExpress from F1 Technologies (www.f1tech.com), written by Mike and Toni Feltman.

SFMenu

This class represents the menu bar. It's a subclass of SFCollection (in SFCOLLECTION.VCX); I discussed that class, which manages collections of objects, in my July 1998 column. SFMenu has only one custom property, cInstanceName, which contains the name of the variable this class is instantiated into (we'll see why we need this later). Although it's public, it has an assign method that makes it read-only to everything except methods of this class. When you instantiate SFMenu, pass the name of the variable as a parameter; the Init method sets cInstanceName to the parameter value. Here's an example:

```
loMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'loMenu')
```

The AddPad method is used to add a new pad to the menu. Pass the class for the pad, the library the class is contained in, and the name to assign to the pad; I suggest using the prompt with "Pad" as the suffix for the name (for example, "FilePad"). The class is instantiated as a member of SFMenu and is also added to the collection so pads can easily be enumerated. Here's an example that calls this method:

```
loMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')
```

You can now reference the new pad as loMenu.EditPad.

The Show method displays the menu. If no pads have been added to the menu, Show calls the DefineMenu method. That method is an abstract method in this class, but in a subclass, you could put a series of AddPad calls that add the desired pads to the menu; this makes the menu subclass self-contained (no outside code has to add pads to the menu). The code then calls the Show method of each pad in the menu and turns on menu handling. Here's the code for Show:

```
local lnI, ;
  loPad
with This
  set sysmenu to
  if .Count = 0
    .DefineMenu()
  endif .Count = 0
  for lnI = 1 to .Count
    loPad = .Item(lnI)
    loPad.Show()
  next lnI
  set sysmenu automatic
endwith
```

The Refresh method refreshes the menu by activating it again; this is useful after you've displayed the menu, and then changed some of the bars or pads in the menu. The ReleaseMembers method, called when the object is destroyed (this is actually defined in SFCustom, the parent class of SFCollection), restores the VFP system menu bar.

SFPad

SFPad is the parent class for all pads in a menu. It too is a subclass of SFCollection. Set the cCaption property to the prompt for the pad (such as "File"), cKey to the hot key (such as "Alt+F"), and cStatusBarText to the message to display in the status bar (such as "Performs file functions"). You can also set cSkipFor to an expression that, when it evaluates to .T., disables the pad, lVisible to .F. if the pad shouldn't currently be visible (you can later set it to .T. to display the pad), and lMRU to .T. if the pad should have an MRU bar in VFP 7 (this property is ignored in VFP 6). Wait a minute: isn't MRU a property of a bar instead of a pad? Yes, but if you look at Office 2000, you'll see that no pad's popup has more than one MRU bar in it, and it's always the last bar. That means that really, MRU status should be a property of the pad (or popup) rather than an individual bar. When lMRU is .T., SFPad will automatically

create an MRU bar using the class specified in the cMRUBarClass and cMRUBarLibrary (by default, "SFMRUBar" and "SFMenu.vcx", respectively).

Notice that in VFP menus, there's no direct relationship between pads and bars. Instead, bars belong to a popup and a popup is associated with a pad. I decided to simplify this in my design; I really couldn't see the need to expose popups at all (if you think about it, you'll realize that's how the VFP Menu Designer works too). So, bars will belong to pads in this design. The Init method automatically creates a popup and stores its name in the protected cPopupName property.

To add a bar to a pad, call the AddBar method, passing the class for the bar, the library the class is contained in, the name to assign to the bar (I suggest concatenating the pad prompt, the bar prompt, and "Bar" for the name, such as "FileExitBar"), and optionally the bar number (if you don't pass this, AddBar will automatically assign the next available bar number to it). The class is instantiated as a member of SFPad and is also added to the collection so bars can easily be enumerated. The Init method of the bar class is passed the name of the popup, the bar number, and .T. if the bar number was specified or .F. if AddBar assigned it. Here's an example that calls AddBar:

```
loMenu.FilePad.AddBar('SFBar', 'SFMenu.vcx', 'FileOpenBar')
```

You can now reference the new bar as loMenu.FilePad.FileOpenBar. To add a separator bar, call the AddSeparatorBar method (it doesn't accept any parameters).

The Show method displays the pad, its popup, and the bars in the popup. If the pad hasn't been defined yet (the protected lDefined property is .F.), the Define method is called to define the pad and create an MRU bar if the lMRU property is .T. Define also calls AddBars. That method is an abstract method in this class, but in a subclass, you could put a series of AddBar calls that add the desired bars to the pad; this makes the pad subclass self-contained. The Show method then calls the either Show or Hide method of each bar in the pad (Hide if the bar is inverted, Show if not). Since this method is called from the Show method of SFMenu, you probably won't have to call this method directly unless you call the Hide method (discussed next). Here's the code for Show:

```
local lnI, ;
  loBar
with This
  if not .lDefined
    .Define()
  endif not .lDefined
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Hide()
    else
      loBar.Show()
    endif loBar.lInvert
  next lnI
endwith
```

The Hide method releases the pad and popup that underlay this class, and sets the lVisible and lDefined properties to .F. You can call this method to make a pad disappear, and then later call Show to make it reappear. Alternatively, you can set lVisible to .F. and then later .T.; this property has an assign method that calls either Hide or Show, depending on the value you're setting it to.

The MRUSelected method is public only because it's called when the MRU bar belonging to the pad is selected. Selecting an MRU bar causes all inverted bars to be displayed, so this method starts by hiding the MRU bar and showing all inverted bars. It then reactivates the popup, and after a selection has been made, redisplays the MRU bar and hides all inverted bars. Here's the code for this method:

```
local lnI, ;
  loBar, ;
  lcPopupName
with This

* Hide the MRU bar, then show all inverted bars.
```

```
  .MRUBar.Hide()
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Show()
    endif loBar.lInvert
  next lnI

* Display the popup again.

  lcPopupName = .cPopupName
  activate popup &lcPopupName

* Now that the popup is closed, show the MRU bar and
* hide all inverted bars.

  .MRUBar.Show()
  for lnI = 1 to .Count
    loBar = .Item(lnI)
    if loBar.lInvert
      loBar.Hide()
    endif loBar.lInvert
  next lnI
endwith
```

Note that inverted bars are only displayed when the user selects the MRU bar, and are hidden again afterward. Other applications that use this technology, such as Word and the Start menu in Windows 2000, typically use a more complex behavior. For example, in Word, if you select an inverted bar, it stays visible until you close the application. If you select the bar again and again over time, it becomes more permanently visible. Bars that originally weren't inverted may become so if you don't use them over time. Obviously, this requires a lot more bar management than the classes I created will support!

The ReleaseMembers method, called when the object is destroyed, calls Hide to destroy the pad and popup.

SFEditPad is a subclass of SFPad. Its AddBars method adds bars (using the SFEditBar class to be discussed later) for Undo, Redo, Cut, Copy, Paste, Clear, and Select All functions. I figured since every application has one of these pads, why bother reinventing the wheel each time?

SFBar

SFBar is the parent class for all bars in a menu. It's based on SFCustom (defined in SFCTRLS.VCX), a subclass of the Custom base class. Set the cCaption property to the prompt for the bar (such as "Open…"), cKey and cKeyText to the hot key and the text for it (such as "Alt+F" for both), and cStatusBarText to the message to display in the status bar (such as "Open a document"). If you want to use a VFP system bar, set cSystemBar to the name of the bar (for example, the SFHelpTopicsBar subclass has cSystemBar set to "_MST_HPSCH" so it automatically has the functionality of that system bar). To conditionally disable the bar, you can either set cSkipFor to an expression or put code in the Allow method that returns .T. if the user is allowed to select the bar. You can also set lEnabled to .F. to unconditionally disable the bar. Set lVisible to .F. if the bar shouldn't currently be visible (you can later set it to .T. to display the bar). To display a mark (such as a checkmark) beside the bar, set lMarked to .T.

There are four ways you can define what happens when the user selects the bar: set cActiveFormMethod to the name of the method of _screen.ActiveForm to call (such as "Find"; you can specify parameters in parentheses if necessary), set cAppObjectMethod to the name of the method of oApp to call (if you use an application object and it's instantiated into a global variable called oApp), set cOnClickCommand to the VFP command to execute, or put the code to execute in the Click method of a subclass of SFBar.

In VFP 7, bars can have graphics; to use this feature, either set cPictureFile to the name of the graphic file to use or cPictureResource to the name of the VFP system bar whose graphic you want to use (for example, "_MFI_NEW" to use the image of the File New bar). VFP 7 also supports inverted bars, so set lInvert to .T. to have the bar only visible when the MRU bar is selected and to have it appear inverted. These properties are ignored in VFP 6.

You likely won't need to call any SFBar methods directly. Show displays the bar by first calling the protected FindBarPosition method if necessary (if the bar number was specified rather than automatically assigned or if the bar is inverted, it may need to be placed between other bars, so FindBarPosition figures out where it should go), and then calling the protected Define method to set up and execute the DEFINE BAR and ON SELECTION BAR commands. The Hide method releases the bar so it disappears from the menu.

By the way, the reason we need to know the name of the variable SFMenu is instantiated into is the ON SELECTION BAR command. We can't use code like "This.Click", because "This" isn't available in the context of a menu selection. Instead, we need to use something like "loMenu.FilePad.FileOpenBar.Click". We can get "FilePad" from This.Parent.Name and "FileOpenBar" from This.Name, but we can't get "loMenu" from any native property. Thus, we need to store the name of the variable the menu was instantiated into in SFMenu.cInstanceName. Then, the entire path to the command to execute can be determined with:

```
This.Parent.Parent.cInstanceName + '.' + ;
  This.Parent.Name + '.' + This.Name + '.Click()'
```

I created a few commonly used subclasses of SFBar. SFHelpTopicsBar (discussed earlier) provides a Help Topics bar. SFSeparatorBar is used when you call SFPad.AddSeparatorBar; it simply has cCaption set to "\-". SFMRUBar is the class used for an MRU bar when you set SFPad.lMRU to .T. SFEditBar is the class used for the bars that SFEditPad (the edit pad class described earlier) creates.

Examples

Let's look at several examples of menus created with these classes. The first, DEMOMENU1.PRG in this month's Subscriber Downloads, defines a menu programmatically using the base menu classes rather than subclasses (one subclass, SFEditPad, is used). SFMenu is instantiated into oMenu, which is declared public so it doesn't go out of scope when the program ends. Two pads, File and Edit, are added to it. New, Open, and Exit bars are added to the File pad; the first two simply display a messagebox when you select them, while the latter terminates the demo menu. Run DEMOMENU1.PRG, try out the menu items, then either choose Exit from the File menu or type oMenu.Release() in the Command window to restore the VFP system menu.

```
public oMenu
oMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'oMenu')

* Create the File pad.

oMenu.AddPad('SFPad', 'SFMenu.vcx', 'FilePad')
with oMenu.FilePad
  .cCaption       = '\<File'
  .cKey           = 'ALT+F'
  .cStatusBarText = 'Performs file functions'

  .AddBar('SFBar', 'SFMenu.vcx', 'FileNew')
  with .FileNew
    .cCaption        = '\<New...'
    .cKey            = 'CTRL+N'
    .cKeyText        = 'Ctrl+N'
    .cStatusBarText  = 'Create a file'
    .cOnClickCommand = [messagebox('You chose ] + ;
      [File, New')]
  endwith

  .AddBar('SFBar', 'SFMenu.vcx', 'FileOpen')
  with .FileOpen
    .cCaption        = '\<Open...'
    .cKey            = 'CTRL+O'
    .cKeyText        = 'Ctrl+O'
    .cStatusBarText  = 'Open a file'
    .cOnClickCommand = [messagebox('You chose ] + ;
      [File, Open')]
  endwith
```

```
  .AddSeparatorBar()

  .AddBar('SFBar', 'SFMenu.vcx', 'FileExit')
  with .FileExit
    .cCaption        = 'E\<xit'
    .cStatusBarText  = 'Restore the VFP menu'
    .cOnClickCommand = 'oMenu.Release()'
  endwith
endwith

* Create the edit pad.

oMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')

* Display the menu.

oMenu.Show()
```

Although the menu it displays is exactly the same as the previous example, DEMOMENU2.PRG is much simpler:

```
public oMenu
oMenu = newobject('MyMenu', 'MyMenu.vcx', '', 'oMenu')
oMenu.Show()
```

That's because all of the menu, pad, and bar definitions are done in subclasses of SFMenu, SFPad, and SFBar. The DefineMenu method of the SFMenu subclass MyMenu adds the MyFilePad and SFEditPad classes to the menu. The AddBars method of the SFPad subclass MyFilePad adds the MyFileNewBar, MyFileOpenBar, and MyFileExitBar classes to the File pad. MyFileNewBar, MyFileOpenBar, and MyFileExitBar are subclasses of SFBar that specify the desired properties and behavior of these bars. Essentially, these subclasses simply do in the Class Designer what DEMOMENU1.PRG did in code. The benefit of using subclasses, though, is that they're now reusable (not that I'd want to use these particular bars anywhere but a demo, but you get the idea).

The third example, DEMOMENU3.PRG, is similar to DEMOMENU1.PRG, but it shows the menu enhancements in VFP 7. The New and Open bars have their cPictureResource properties set to _MFI_NEW and _MFI_OPEN, respectively, and Exit has cPictureFile set to CLOSE.BMP, so these bars will display graphics. The File pad's lMRU property is set to .T. and an inverted bar, Print Setup, is added between the Open and Exit bars. When you run DEMOMENU3.PRG, you won't see the Print Setup bar initially. Click on the MRU bar at the bottom of the menu, and Print Setup appears. After you choose an item from the menu (other than Exit, of course), Print Setup disappears again.

Conclusion

Object-oriented menus have long been a requested enhancement to VFP. However, as you can see from the simple classes presented here, they're easy to implement right now. These menu classes will make it easier to create reusable menus for your applications. Although I didn't create one, a great add-on for these classes would be a visual tool to create a menu. This might be a GENMENUX driver for the VFP Menu Designer that generates the proper code or classes, or a completely new tool.

Another idea is a data-driven menu. A routine would read the pad and bar information from a table and create the menu programmatically similar to how DEMOMENU1.PRG created its menu. Changing the menu would be as easy as changing the table; you might even allow your end-users to customize the menus for themselves by simply added, editing, or deleting records in the table. You could even use XML rather than a table. The ideas are endless!

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), co-author of "What's New in Visual FoxPro 7.0" from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com Email: dhennig@stonefield.com