PEMs For Your Base Classes

Doug Hennig

This month's column examines some ideas to put in your subclasses of Visual FoxPro base classes. Some of these ideas are straightforward, but others may be less obvious. We'll look at some custom properties and methods you can add to provide more automatic functionality to your applications.

You've probably heard a thousand times by now that using the Visual FoxPro base classes in your applications is bad, and that you should create your own subclass of each VFP base class. I won't bore you with the reasons why in this article. Instead, we'll take a look at the subclasses I've created for my use. In this article, I refer to the subclasses I've created as "my base classes" because I always use them in my applications as the starting point for any subclasses. When I want to refer to a class built into VFP (such as a ComboBox), I'll refer to it as a "VFP base class".

While you could create a set of your own base classes by simply subclassing VFP base classes without making any changes, I find that I usually end up changing or adding something to any VFP base class, so why don't we make our base classes have the behavior we want directly? We'll look at the changes I've made to the properties, events, and methods (PEMs) of VFP's classes. Some of these changes are straightforward, such as setting the BackStyle property of a Label subclass to Transparent so the background of the container shows through. Others may be less obvious, such as having the Error method of an object look up its containership hierarchy to find the first parent with error handling code.

Common PEMs

All classes in the CONTROLS.VCX class library (on the Developer disk) have some common properties and methods added or filled in. In addition, some classes that have common behavior (such as those that allow the user to change their value) have the same methods added or filled in. These common PEM changes are described here.

About

This custom method is added to every class as a way to document the class. I've seen other developers add "Copyright" or "Documentation" methods to provide the same purpose. Whatever the name of the method, it generally just consists of comments that describe the features of the class. Some folks enter the comment text between TEXT and ENDTEXT statements (so you don't have to comment out each line); I just used comment lines. In the About method for each class, I put the following information:

- the name of the class
- the class this class is based on
- the purpose of the class
- the author, copyright, and last revision date
- the name of the INCLUDE file used by the class if any
- changes made to properties of the parent class

- changes made to methods of the parent class
- custom public properties added to this class
- custom protected properties added to this class
- custom public methods added to this class
- custom protected methods added to this class

Here's the About method from the SFCheckBox class as an example:

```
*=================================================
* Class:          SFCheckBox
* Based On:       CheckBox
* Purpose:        Base class for all CheckBox objects
* Author:         Doug Hennig
* Copyright:      (c) 1996 Stonefield Systems Group Inc.
* Last revision: 11/02/96
* Include file:  none
*
* Changes in "Based On" class properties:
* AutoSize:       .T.
* BackStyle:      0 (Transparent)
* Value:          .F. since checkboxes usually are
*                 used for logical values
*
* Changes in "Based On" class methods:
* Error:          calls the parent Error method so
*                 error handling goes up the
*                 containership hierarchy
* InterActiveChange: call AnyChange()
* ProgrammaticChange:  call AnyChange()
*
* Custom public properties added:
* Builder:        holds the name of a custom builder
*
* Custom protected properties added:
* None
*
* Custom public methods added:
* About:          for documentation purposes
* AnyChange:      called by InteractiveChange() and
*                 ProgrammaticChange()
* Release:        releases the object
*
* Custom protected methods added:
* None
*=================================================
```

Error

In the June and July 1996 issues of FoxTalk, I discussed an error handling strategy in which all objects handle any errors they can and pass any other errors to the Error method of the form they reside in so common error handling services can be centralized in one place. I've since refined this scheme; objects now call the Error method of their container rather than jumping directly up to the form (actually, calls go up the class hierarchy first before going up the containership hierarchy). This allows additional levels of localized error handling to exist. However, this scheme has one problem: controls sitting on base class Page, Column, or other containers with no Error method code essentially have no error trapping because they call an empty method! The solution is to travel up the containership hierarchy until we find a parent that has code in its Error method. If we can't find such a parent, then display a generic error message (this isn't likely, since I

base all forms on the SFForm class, which does have Error method code). Here's the code used in all objects except SFForm and SFToolbar:

```
LPARAMETERS nError, cMethod, nLine
local oParent
oParent = iif(pemstatus(Thisform, ;
  'FindErrorHandler', 5), ;
  Thisform.FindErrorHandler(This), .NULL.)
if isnull(oParent)
  = messagebox('Error #' + ltrim(str(nError)) + ;
    ' occurred in line ' + ltrim(str(nLine)) + ;
    ' of ' + cMethod, 0, _VFP.Caption)
else
  oParent.Error(nError, This.Name + '.' + cMethod, ;
    nLine)
endif isnull(oParent)
```

This code checks to see if the form the control is sitting on has a FindErrorHandler method, and if so, calls it to locate the first parent of the control with code in its Error method (we'll see the code for this method in a moment). If such a parent is found, its Error method is called with the same parameters that this Error method received, except the name of the object is added to cMethod so our error handling services can know in which object the error originated.

Because they are the "top-level" containers (I never use form sets), the Error method for the SFForm and SFToolbar classes are different than other objects. This method checks to see if an oError object has been added to the form, and if so, calls its ErrorHandler method to provide global error handling services. If not, an error message is displayed. It's beyond the scope of this article to discuss an error handling object; see my column in the July 1996 issue for a simple example of such a class.

```
LPARAMETERS nError, cMethod, nLine
local laError[1]
aerror(laError)
if type('This.oError') = 'O' and ;
  not isnull(This.oError)
  This.oError.ErrorHandler(nError, ;
    This.Name + '.' + cMethod, nLine)
else
  if messagebox('Error #' + ltrim(str(nError)) + ;
    ' (' + laError[2] + ')' + chr(13) + ;
    'occurred in line ' + ltrim(str(nLine)) + ;
    ' of ' + cMethod, 17, _VFP.Caption) = 2
    cancel
  endif messagebox ...
endif type('This.oError') = 'O' ...
```

As I mentioned above, the Error method of all objects call the FindErrorHandler method of the form. This method accepts as a parameter a reference to the object that called it. It starts with the parent of that object and travels up the containership hierarchy until it finds a parent with code in its Error method. This prevents the problem of error handling stopping on base class Page, Column, or other containers because they have no code in the Error method. Here's the code for that method:

```
lparameters oObject
local oParent
oParent = oObject.Parent
do while type('oParent') = 'O' and ;
```

```
  not isnull(oParent)
  do case
    case pemstatus(oParent, 'Error', 0)
      exit
    case type('oParent.Parent') = 'O'
      oParent = oParent.Parent
    otherwise
      oParent = .NULL.
  endcase
enddo while type('oParent') = 'O' ...
return oParent
```

Builder

Ken Levy created a tool for VFP 3 called BuilderX that was included with the *Visual FoxPro 3 Codebook* by Yair Alan Griver. This tool acts a wrapper for VFP's native builder application (BUILDER.APP). Its main purpose is to allow you to define a specific builder for each class, stored in a custom Builder property of the class. When you invoke the builder for an object created from the class, BuilderX first checks if the object has a Builder property and if it contains anything. If so, it instantiates an object of the class specified in that property (you can also specify the class library your builder class is located in by entering the VCX name and class name separated by a comma). If not, it calls BUILDER.APP to carry on as it normally would. This allows you to create customized builders for any class (perhaps using Ken's BuilderB tool, which is the source of another article) and specify that builder in the Builder property of the class itself. This idea is so cool that Microsoft added this behavior to the native BUILDER.APP in VFP 5 so you don't need BuilderX.

I've added a custom Builder property to all my base classes so I can take advantage of this capability. This property contains a blank value for all classes, so it's really there as an "abstract" property. I use it for specialized subclasses of my base classes, although you could also use it for simple builders for even base classes.

Release

I added a custom Release method to any class which doesn't already have one. It contains just one line of code:

```
release This
```

This permits you to use a common way to release any object: <object>.Release().

AnyChange, InteractiveChange, ProgrammaticChange

Classes that allow the user to change their values (such as TextBox, CheckBox, EditBox, etc.) have InteractiveChange and ProgrammaticChange methods. Usually, if you put any code into one of these methods, you end up having the other method call it so the same behavior takes place regardless of how the object's value was changed. To make this work automatically, I added a custom AnyChange method to those classes and added the following to their InteractiveChange and ProgrammaticChange methods:

```
This.AnyChange()
```

Thus, any code needed when any change is made to the value of the control should be entered into the control's AnyChange method.

SelectOnEntry

Microsoft added this new property to several controls in VFP 5. If you set it to .T., the text in the control will automatically be selected when the control receives focus. Since this is the default behavior I want, I set it to .T. in all classes with this property.

Valid, Validation

The Valid method of controls that have it (such as ComboBox and TextBox) has two things that annoy me:

- It's fired even when the user clicks on a "cancel changes" button (unless this button is in a toolbar), causing the goofy behavior that the user has to enter a valid value into a control before they can cancel the changes made to a record.
- If I put code into the Valid method of a class (for example, to overcome the first problem), when I need custom Valid code in a subclass or an object instantiated from the class, I usually have to do something like:

```
dodefault()  && execute the normal behavior
* custom validation code here
nodefault  && don't re-execute the normal behavior
```

To overcome the first problem, I added a custom lCancel property to my CommandButton base class, which allows me to indicate that a button is a "cancel changes" button (the Cancel property of a button indicates that the button is selected by pressing Escape, but I may not want that behavior for my "cancel changes" button). The Valid method of all classes that have one then checks to see if a "cancel changes" button was pressed before performing any validation code. It does this by using the SYS(1270) function to get an object reference to the control the user just clicked on and checking to see if that object has an lCancel property and if so, whether it's .T. or not.

To solve the second problem, I added a new Validation method to these same classes and made the Valid method call it. This way, I leave the Valid method of an object alone and instead put my specific validation code into the Validation method.

Here's the code for the Valid method:

```
local oObject
oObject = sys(1270)
if lastkey() = 27 or (type('oObject') = 'O' and ;
  type('oObject.lCancel') = 'L' and oObject.lCancel)
  return .T.
endif lastkey() = 27 ...
return This.Validation()
```

My Base Classes

Here's the description of the PEM changes in each of my base classes. These classes can be found in the CONTROLS.VCX file on the Developer disk. All classes are named according to the VFP base class they're based on plus an SF (for Stonefield) prefix. For example, SFCheckBox is based on the CheckBox base class.

SFCheckBox

AutoSize: set to .T. so I don't have to manually size the control to fit the caption.
BackStyle: set to 0-Transparent so the background color of the container shows through.
Value: set to .F. because I normally use a CheckBox for logical values.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFComboBox

aItems[1]: I added this custom property so if desired, the ComboBox could be self-contained; its source of display items is a property of itself.
BoundTo: I set this new VFP 5 property to .T. so the ComboBox will properly work with numeric data sources (see the VFP 5 documentation for more information on this property).
GetActiveProperties, *Init*, and *lActive*: see SFSpinner for information on these changes. Init also initializes the This.aItems array to an empty string.
ItemTips: set to .T. so this new VFP 5 feature is turned on.
RowSource and *RowSourceType*: set to This.aItems and 5-Array, respectively, since I most frequently have ComboBoxes tied to arrays. You can, of course, easily override these properties as needed.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFCommandButton

lCancel: .T. if this button is used as a "cancel" button (this allows the Valid method of a control to not bother doing validation if the user clicked a "cancel changes" button as described earlier).
About, *Builder*, *Error*, and *Release*: described earlier.

SFCommandGroup

BackStyle: set to 0-Transparent so the background color of the container shows through.
ButtonCount: 0 so buttons can be added from the SFCommandButton class if desired.
About, *Builder*, *Error*, and *Release*: described earlier.

SFContainer

BackStyle and *BorderWidth*: set to 0-Transparent and 0, respectively, so the container itself has no visible appearance.
SetEnabled: while disabling a container causes the member objects of the container to be disabled, they don't appear to be disabled. I added this custom method to set the Enabled property of the object and all member objects to the specified value so all objects appear to be enabled or disabled appropriately:

```
lparameters tlEnabled
This.SetAll('Enabled', tlEnabled)
This.Enabled = tlEnabled
```

About, *Builder*, *Error*, and *Release*: described earlier.

SFControl

BackStyle and *BorderWidth*: set to 0-Transparent and 0, respectively, so the control itself has no visible appearance.
SetEnabled: see SFContainer.
About, *Builder*, *Error*, and *Release*: described earlier.

SFCustom

About, *Builder*, *Error*, and *Release*: described earlier.

SFEditBox

IntegralHeight: set to .T. so the box is always displays the last line of text properly.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFForm

Destroy: hides the form so it appears to go away faster:

```
This.Hide()
```

Init: because the AutoCenter and BorderStyle properties are used at design time as well as run time, and they can be a hassle when used at design time, I added new lAutoCenter and nBorderStyle properties to contain the desired run time AutoCenter and BorderStyle values, and made the Init form do the following:

```
with This
  .AutoCenter  = .lAutoCenter
  .BorderStyle = .nBorderStyle
endwith
```

lAutoCenter and *nBorderStyle*: new custom properties as described above. They default to .T. and 2 (double border), respectively.
Load: set some environmental things the way we want:

```
set talk off
set safety off
set deleted on
set fullpath on
set exact off
set unique off
```

oError: a reference to an ErrorMgr object.
ShowTips: set to .T. so tool tips appear.
About, *Builder*, *FindErrorHandler*, and *Error*: described earlier.

SFGrid

AllowHeaderSizing, *AllowRowSizing*, and *SplitBar*: all set to .F. since I don't want that behavior by default.

About, *Builder*, *Error*, and *Release*: described earlier.

SFImage
BackStyle: set to 0-Transparent so the background color of the container shows through.
About, *Builder*, *Error*, and *Release*: described earlier.

SFLabel
AutoSize: set to .T. so I don't have to manually size the control to fit the caption.
BackStyle: set to 0-Transparent so the background color of the container shows through
About, *Builder*, *Error*, and *Release*: described earlier.

SFLine
About, *Builder*, *Error*, and *Release*: described earlier.

SFListBox
aItems[1]: I added this custom property so if desired, the ListBox could be self-contained; its source of display items is a property of itself.
IntegralHeight: set to .T. so the box is always displays the last line of text properly.
ItemTips: set to .T. so this new VFP 5 feature is turned on.
RowSource and *RowSourceType*: set to This.aItems and 5-Array, respectively, since I most frequently have ListBoxes tied to arrays. You can, of course, easily override these properties as needed.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFOLEBoundControl
About, *Builder*, *Error*, and *Release*: described earlier

SFOptionButton
AutoSize: set to .T. so I don't have to manually size the control to fit the caption.
BackStyle: set to 0-Transparent so the background color of the container shows through.
About, *Builder*, *Error*, and *Release*: described earlier.

SFOptionGroup
BackStyle: set to 0-Transparent so the background color of the container shows through.
ButtonCount: 0 so buttons can be added from the SFOptionButton class if desired.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, and *Release*: described earlier.

SFPageFrame
PageCount: 0 so pages can be added or removed more easily.
TabStyle: I set this new VFP 5 property to 1-Nonjustified because I want this appearance by default.
About, *Builder*, *Error*, and *Release*: described earlier.

SFSeparator

About and *Release*: described earlier.

SFShape

BackStyle: set to 0-Transparent so the background color of the container shows through.
SpecialEffect: set to 0-3D.
About, *Builder*, *Error*, and *Release*: described earlier.

SFSpinner

GetActiveProperties: if the custom property lActive is .T., this method sets the InputMask and Format properties of the control to the values stored in the database for the control's ControlSource. This allows these properties to be updated dynamically whenever the database properties change. Here's the code for this method:

```
local lcControl, ;
  lnDot, ;
  lcAlias, ;
  lcFormat, ;
  lcInput
with This
  lcControl = .ControlSource
  lnDot     = at('.', lcControl)
  if .lActive and lnDot > 0 and not empty(dbc())
    lcAlias = left(lcControl, lnDot - 1)
    if indbc(lcAlias, 'Table') or ;
      indbc(lcAlias, 'View')
      lcFormat   = dbgetprop(lcControl, 'Field', ;
        'Format')
      lcInput    = dbgetprop(lcControl, 'Field', ;
        'InputMask')
      .Format    = iif(empty(lcFormat), .Format, ;
        lcFormat)
      .InputMask = iif(empty(lcInput),  .InputMask, ;
        lcInput)
    endif indbc(lcAlias, 'Table') ...
  endif .lActive ...
endwith
```

Init: if the custom property lActive is .T., get the active properties for this control by calling the GetActiveProperties method.

```
with This
  if .lActive
    .GetActiveProperties()
  endif .lActive
endwith
```

lActive: this custom property, which is initially set to .F., indicates whether the Init method calls the custom GetActiveProperties method.
About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFTextBox

GetActiveProperties, *Init*, and *lActive*: see SFSpinner.

About, *AnyChange*, *Builder*, *Error*, *InteractiveChange*, *ProgrammaticChange*, *Release*, *SelectOnEntry*, *Valid*, and *Validation*: described earlier.

SFTimer

About, *Builder*, *Error*, and *Release*: described earlier.

SFToolbar

Destroy: hides the toolbar so it appears to go away faster:

```
This.Hide()
```

oError: a reference to an ErrorMgr object.
About, *Builder*, *FindErrorHandler*, *Error*, and *Release*: described earlier.

Conclusion

Some of the ideas for the PEM changes I discussed in this article came from other folks, especially the FoxGang on CompuServe, who every day unselfishly share a wealth of great ideas. I don't recall (or even know) who originated some of these ideas, but if you recognize your idea in this column, thank you!

Next month, we'll take a look at some special subclasses of these base classes, including an EditBox that automatically expands keywords to complete text (similar to the AutoCorrect function in WinWord) and a ComboBox that supports a feature similar to Quicken's "quick fill" function.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., 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. CompuServe 75156,2326.