Mining for Gold in the FFC

Doug Hennig

Introduction

One of the design goals for VFP 6 was to make it easier for programmers new to VFP to get up and running with the tool. The FoxPro Foundation Classes, or FFC, is one of the results of this goal. The FFC, located in the FFC subdirectory of the VFP home directory, is a collection of class libraries that provide a wide range of functions. Don't think that just because previous versions of FoxPro have included some, shall we say, less useful (to be polite) example files that these fall into that category. While some of these do appear to be more demoware than really useful, there are still lots of great classes in here. It's well worth the effort to spend some time looking at these classes, picking through the rocks to find the nuggets.

The best way to check out the FFC is using a new VFP 6 tool called the Component Gallery, accessible from the VFP Tools menu. I won't discuss the features of the Component Gallery in this document; it's simple enough to use that you can navigate your way around just by playing with it. The FFC classes are displayed in the Foundation Classes folder of the Visual FoxPro Catalog (when I refer to where classes can be found later in this document, I won't specify this folder or this catalog, just the subfolder under Foundation Classes). Classes are grouped by type (for example, Buttons, Dialogs, and Utilities) and display a description in the status panel when you select them. Even better, right-clicking on a class displays a context menu giving you access to the Help topic and sample files (either to run or to view) for that class. This makes it very easy to look through the FFC and see which classes might interest you.

This document will look at some of the FFC classes and see how we might use them or even subclass them to make them even more useful. Before we get started, though, let's talk about _BASE.VCX.

Base Classes

VFP 6 includes a set of subclasses of VFP base classes in _BASE.VCX. Although these classes aren't located in the Foundation Classes folder in the Component Gallery (they're in the My Base Classes folder), this VCX is located in the FFC subdirectory and all FFC classes are subclassed from _BASE classes. If we use FFC classes in our applications, _BASE.VCX comes along for the ride. So, a question arises: although you probably have your own set of base classes, should you consider using _BASE classes for your base classes instead? The reason for even considering this is why have two VCXs in your projects that provide essentially the same thing?

After looking at the _BASE classes, the conclusion I've come to is no; I'll continue using my own SFCTRLS.VCX classes. _BASE classes don't have the visual changes I've made to my base classes; for example, the AutoSize property for classes such as _CheckBox and _Label is set at the default .F.; I almost always want it set to .T., so that's what I've done in SFCheckBox, SFLabel, and other classes with this property. _BASE classes also don't have the behavior I want; for example, their Error methods pass the error on to an ON ERROR handler rather than using the Chain of Responsibility design pattern (up the class hierarchy and then the containership hierarchy) my classes do as I discuss in my Error Handling document (available from the Technical Papers page of www.stonefield.com). Finally, these classes have a lot of custom properties and methods that support the VFP Application Wizard. Since I don't plan to use the Application Wizard to create applications, these properties and methods would just complicate things. So, we'll just have to live with the fact that _BASE.VCX will be included in our projects, whether we want it or not, if we use FFC classes.

_SysToolbars

It's highly unlikely you'll want the VFP development environment toolbars (such as the Standard and Database toolbars) to be visible when an application is running. Unfortunately, there isn't a single command that hides them all, so most developers create an array of toolbar names, spin through the array and see if the current toolbar is visible or not, and if so, hide it. Of course, you have to do just the opposite when the application closes down, at least in a development environment.

Because this is a common task, the FFC includes a class called _SysToolbars; this class is located in _APP.VCX and appears as "System Toolbars" in the Application subfolder of the Component Gallery. This class is very simple; it hides and restores the system toolbars either automatically or manually. To use it automatically, either pass .T. to its Init method when you instantiate it programmatically or set its lAutomatic property to .T. in the Property Sheet when you drop it on a form. In this case, any visible system toolbars are hidden when the object is instantiated and redisplayed when it's destroyed. To use it manually, call its HideSystemToolbars and ShowSystemToolbars methods. You'll likely want to instantiate it as soon as possible at application startup using code similar to:

```
oSysToolbars = newobject('_SysToolbars', '_app.vcx', ;
    '', .T.)
```

As long as oSysToolbars stays in scope, the toolbars stay hidden. You can either manually release this object or let it go out of scope when the application terminates.

_Resizable

This class, which is defined in _CONTROLS.VCX and appears as "Resize Object" in the User Controls subfolder of the Component Gallery, moves and resizes all the controls in a form when the form is resized. It takes care of all the mundane details of drilling down through containers (such as PageFrames), adjusting the Top, Left, Height, and Width properties of controls. I've done this kind of thing manually before in the Resize method of the form, writing reams of code to handle all the applicable controls. Believe me, anything that can automate this tedious chore is very welcome.

To see how _Resizable works, run the ResizeDemo form (by the way, this form also contains an _SysToolbars object with lAutomatic set to .T. so you can see how that class works). Leave “Resize” unchecked and resize the form. See how dumb it looks as more background appears when you make the form larger or controls get hidden as you make the form smaller? Not the sign of a professional application. Now check “Resize” (but leave “SFResizable” unchecked for now); when you resize the form, the editbox is resized and the other controls are moved automatically.

However, there are a couple of problems with this class. First, it resizes or moves all the controls. Typically, when you resize a form, you want to resize certain controls (such as editboxes) and move the ones below and to the right of them to account for the new size; the rest you want to leave alone. Another problem is that it moves everything proportional to the original size of the form; the effect is that a resized form looks like a blown up or shrunken version of the original, with controls further apart or close together. The result is sort of like what happens to writing on a balloon as it’s blown up. In my experience, what you really want is a form that looks just like it was but with some of the controls larger or smaller and the rest just moved relative to the resized controls. After all, the whole reason for the user to resize a form is to be able to see more of those controls they’d normally have to scroll (grids, listboxes, editboxes, TreeView and ListView controls, etc.), not to get a bigger form with more background.

The good news is that we don’t need to throw _Resizable out and create a new class with the behavior we want. _Resizable has all of the logic we need; it just doesn’t do things quite right. So, let’s subclass it and make the subclass act the way we expect.

The subclass I created is called SFResizable, and it’s contained in SFFFC.VCX. I changed both the Height and Width properties to 17 so the control doesn’t take up as much space when it’s dropped on a form. Since we need to treat some controls differently than others (some will be moved while others may be resized), we need a way to indicate how each control should be treated. I originally considered adding new properties to my base classes (for example, lResize, which, if .T., would indicate that this control should be resized) but rejected that because then SFResizable would only work with a certain set of base classes, and that would make it far less reusable. Instead, I decided to make SFResizable self-contained, so I created the following new properties:

- cRepositionLeftList: a comma-delimited list of the names of controls that should be moved left-right only as the form is resized.

- cRepositionList: a comma-delimited list of the names of controls that should be moved left-right and up-down as the form is resized.

- cRepositionTopList: a comma-delimited list of the names of controls that should be moved up-down only as the form is resized.

- cResizeHeightList: a comma-delimited list of the names of controls that should be resized taller or shorter only as the form is resized.

- cResizeList: a comma-delimited list of the names of controls that should be resized as the form is resized.
- cResizeWidthList: a comma-delimited list of the names of controls that should be resized wider or narrower only as the form is resized.

I also overrode the AddToArray and SetSize methods. AddToArray adds size and position information about a control to an array property of the class; it's called from the LoopThroughControls method, which processes all of the controls in the form. The problem with the original AddToArray is that it stores the size and position information of the control as values proportional to the size and position of the form rather than the actual values for the control. So, I simply used the Visual Basic method of subclassing (I copied the code from _Resizable.AddToArray and pasted it into SFResizable's method) and then modified the code to store the original values. SetSize is also called from LoopThroughControls; it adjusts the size and position of a control by the difference between the original (stored in the InitialFormHeight and InitialFormWidth properties) and current sizes of the form. Since I don't want every control resized and moved, I changed the code to use the following logic:

- If the control's name is in the cRepositionList or cRepositionTopList properties, its Top value is adjusted.
- If the control's name is in the cRepositionList or cRepositionLeftList properties, its Left value is adjusted.
- If the control's name is in the cResizeHeightList or cResizeList properties, its Height value is adjusted.
- If the control's name is in the cResizeWidthList or cResizeList properties, its Width value is adjusted.

The Reset method is used to reset the control when the form is resized in some way other than by the user resizing it. An example of this is having a button that "expands" the form, making more controls visible. Under these conditions, _Resizable no longer works correctly because it hasn't taken into account the new size of the form. I overrode the Reset method in SFResizable to reset the InitialFormWidth and InitialFormHeight properties to the current width and height of the form.

To use SFResizable, drop it on a form and call its AdjustControls method in the Resize method of the form. Enter the names of those controls that should be resized for height and width as the form is resized into the cResizeList property of the SFResizable object; grids, editboxes, and other controls that can scroll are obvious candidates. Enter the names of those controls that should only be resized wider or narrower as the form is resized into the cResizeWidthList property. Enter the names of those controls that should only be resized taller or shorter as the form is resized into the cResizeHeightList property; an example might be a listbox that should remain a constant width. Enter the names of those controls that should be moved up-down and left-right as the form is resized into the cRepositionList property. For example, controls below and to the right of an editbox might qualify for this adjustment. For those that should only be moved up and down, enter their names into the cRepositionTop property. Examples include controls below a grid, because these controls need to move up or down as the grid's Height is changed. Finally, enter the names of controls that should be moved left and right, such as those to the right of an editbox, as the form is resized into the cRepositionLeft property.

Run the ResizeDemo form again, but this time check "SFResizable" rather than "Resize". When you resize the form, the textbox beside "Label 1" doesn't move, the editbox and shape surrounding it don't move but are resized, the "Resize" and "SFResizable" checkboxes and the checkboxes beside the editbox move left and right but not up and down, the textbox beside "Label 2" moves up and down but not left and right, and the "Reset" button moves both up-down and left-right. In other words, the form behaves as we'd expect when it's resized.

A perfect addition to SFResizable would be a builder to make it easy to specify which controls should be moved or resized, with perhaps a list of the controls in the form and checkboxes indicating how the selected control should be treated.

Registry

As you know, the Registry is the "in" place to store configuration, preference, and other application settings; INI files are officially passe (although I know lots of developers who, like me, would rather lead a user through editing an INI file over the phone than dare have them use a tool like REGEDIT). Settings should normally be stored in keys with a path similar to HKEY_CURRENT_USER\Software\My Company Name\My Application\Version 1.1\Options.

The FFC Registry class (defined in REGISTRY.VCX and appearing as "Registry Access" in the Utilities subfolder of the Component Gallery) is a wrapper class for the Windows API calls that deal with the Registry. Rather than having to know that you must open a key using the RegOpenKey function before you can use RegQueryValueEx to read the key's value, all you have to know with the Registry class is that you call its GetRegKey method. Here's an example that gets the name of the VFP resource file:

```
#include REGISTRY.H    && located in HOME() + 'FFC'
loRegistry = newobject('Registry', 'Registry.vcx')
lcResource = ''
loRegistry.GetRegKey('ResourceTo', @lcResource, ;
    'Software\Microsoft\VisualFoxPro\6.0\Options', ;
    HKEY_CURRENT_USER)
```

(Yeah, I know it's easier to use SET('RESOURCE', 1), but this is just an example).

REGISTRY.VCX also contains subclasses of Registry that make it easier to read and write VFP settings (FoxReg), ODBC settings (ODBCReg), applications by file extension (FileReg), and even (horrors!) INI files (OldINIReg). Registry-specific constants are defined in REGISTRY.H so you can specify HKEY_CURRENT_USER (as I did above) rather than its value (-2147483647).

Here are the useful methods in Registry. Unless otherwise specified, all return a code indicating success (0, or the constant ERROR_SUCCESS) or the WinAPI error code; see REGISTRY.H for error codes.

- GetRegKey: puts the value of the specified key into a variable.
- SetRegKey: sets the value of the specified key to the specified value (the entire key path is created if it doesn't exist).
- DeleteKey: deletes the specified key (and all subkeys) from the Registry.
- DeleteKeyValue: remove the value from the specified key.
- EnumOptions: populates an array with all the settings under a specific key and their current values.
- IsKey: returns .T. if the specified key exists.

Registry can be instantiated at application startup (perhaps by an application object) to get the settings the application needs: the location of the data files on a LAN, the names of the most recently used forms (so they can appear at the bottom of the File menu like Microsoft applications do), etc. It can also be dropped on a form to save and restore form-specific settings such as the Left, Top, Height, and Width values (so it has the same size and position as when this user closed it), grid column widths and positions (so users can rearrange grids and have them appear that way the next time), etc.

Being a picky guy, I have two problems with the Registry class. First, because the return value of GetRegKey is a success or error code, you have to pass the variable you want the value placed into by reference. I'd prefer it to return the value of the key; after all, if an error occurs, the WinAPI error code is probably as useful to me as the "details" section of a GPF dialog, and even if I do want it, it could easily be stored in a property of Registry I can query. Second, it's likely that both the key path ("Software\Microsoft\VisualFoxPro\6.0\Options" in the example above) and the user key (usually HKEY_CURRENT_USER) are going to be the same for every method call of a specific instance of a Registry object. Being the lazy sort, I'd rather put these values into properties and not pass them every time I call a method.

As with _Resizable, I've subclassed Registry into SFRegistry (also in SFFFC.VCX) to have the behavior I want. Although I planned to, it turned out that I didn't have to add properties for the default key path (cAppKeyPath) and user key (nUserKey); they already existed even though they're not used by Registry (they're used by the subclasses in REGISTRY.VCX). Since Registry.Init sets nUserKey to HKEY_CURRENT_USER, there's isn't even normally a need to change this property. I changed the GetRegKey, SetRegKey, DeleteKey, DeleteKeyValue, EnumOptions, and IsKey methods to accept different parameters than the matching Registry class methods (I rearranged the parameters so optional ones are at the end) and return a more appropriate value (the key value in the case of GetRegKey, the number of options in the array in the case of EnumOptions, and .T. if the method succeeded in the case of SetRegKey, DeleteKey, and

DeleteKeyValue). These methods also store the WinAPI result code into a new nResult property, which can be used to determine what went wrong if a method fails. Here's an example; this is the code from EnumOptions:

```
lparameters taRegOptions, ;
    tlEnumKeys, ;
    tcKeyPath, ;
    tnUserKey
local lcKeyPath, ;
    lnUserKey, ;
    lnReturn

* If the key path and user key weren't passed, use the
* defaults.

lcKeyPath = This.GetKeyPath(tcKeyPath)
lnUserKey = This.GetUserKey(tnUserKey)

* Use the parent class method to enumerate the key,
* store the result code, and return the number of
* options it found.

lnSuccess = dodefault(@taRegOptions, lcKeyPath, ;
    lnUserKey, tlEnumKeys)
This.nResult = lnSuccess
lnReturn = iif(lnSuccess = ERROR_SUCCESS, ;
    alen(taRegOptions, 1), 0)
return lnReturn
```

GetKeyPath and GetUserKey are new methods called by all the overridden methods to either use the key path and user key values passed or the defaults (stored in the cAppPathKey and nUserKey properties) if they weren't passed.

Here's an example of the use of SFRegistry; this code would be used in the Init method of a form to restore the size and position it had the last time the user had it open. This code assumes an SFRegistry object named oRegistry was dropped on the form and its cAppPathKey property was set in the Property Sheet to the key path for this application.

```
lcKey = This.oRegistry.cAppPathKey + '\' + This.Name
lcTop = This.oRegistry.GetRegKey('Top', lcKey)
if not isnull(lcTop)
    This.Top = val(lcTop)
endif not isnull(lcTop)
* similar code for Left, Width, and Height
```

(Since the Registry class only supports reading and writing strings, VAL() must be used on the return value. Also, if this is the first time this form is run, a key may not exist for it in the Registry, in which case .NULL. is returned, so this code handles that case). A method of the form (such as Release) would use This.oRegistry.SetRegKey to save the current form size and position in the Registry.

For another example, run REGISTRYDEMO.PRG. It creates some new keys, displays their values, and shows how EnumOptions works. It deletes the keys it creates so it doesn't pollute your Registry permanently.

_ObjectState

If you've been doing your homework on design patterns, you're probably aware of a pattern known as Memento. A Memento is intended to save the state of something, presumably so it can be restored after it's been changed. The FFC includes a class called _ObjectState that's sort of like a Memento: it saves the value of one or more properties of another object and can later restore them to their former values.

_ObjectState is defined in _APP.VCX and appears as "Object State" in the Application subfolder in the Component Gallery. It has an oObject property that contains an object reference to the object whose state it maintains; this property can be set by passing the object reference to the Init method of the _ObjectState object or by setting it manually. It has a Set method that sets the value of the specified property of the managed object to the specified value, optionally first saving the original value in an array property of itself. It also has a Restore method that restores the value of one or all saved properties. The Restore method is called from the Destroy method, so when the _ObjectState object goes out of scope, everything is restored automatically. Since the array property has a single row for each saved property, you can't use this class to provide multiple levels of undo; however, you could subclass _ObjectState and add this behavior if desired.

Where might we use such a class? One situation involves things a user can change but might want to change back. For example, you might allow a user to rearrange and resize the columns in a grid, but provide a "Reset to Default" function that puts them back. Another place would be when you temporarily change the properties of an object (perhaps so it behaves differently), do something with the object, and then change them back again. Rather than having a set of local variables that save the property values and then having to manually restore them before the routine ends, you could do something like this:

```
local loObjectState
loObjectState = newobject('_ObjectState', '_app.vcx', ;
    '', This)
loObjectState.Set('<first property', <new value>)
loObjectState.Set('<second property', <new value>)
* do something here
```

When this code ends, loObjectState goes out of scope, so the saved property values are automatically restored.

As usual, I have one slight quibble with the class implementation: saving the current value of the property is optional. There really isn't a need to use _ObjectState if you're not going to save the value (after all, you can just store the new value to the property yourself in that case), so I think the default behavior should be to save and have it optionally not save. So, I created a subclass of _ObjectState called SFObjectState (in SFFFC.VCX) that simply overrides the Set method as follows:

```
lparameters tcProperty, ;
    tuValue, ;
    tlNoSave
return dodefault(tcProperty, tuValue, not tlNoSave)
```

In other words, rather than passing .T. to save, you have to pass .T. to not save.

To see an example of this class in action, run the ResizeDemo form, check both “Resize” and “SFResizable”, then resize the form. Now click on the Reset button and watch everything snap back to the original size and position. This was done by saving the Height and Width properties of the form in its Init method:

```
with This.oObjectState
    .oObject = This
    .Set('Height', This.Height)
    .Set('Width',  This.Width)
endwith
```

and restoring them in the Click method of the button:

```
Thisform.oObjectState.Restore()
```

Of course, we didn’t have to save the original size and position of every object on the form because changing the form Height and Width causes the Resize method to fire, which moves everything for us. Also, although I didn’t specify individual properties to Restore (so it restored all saved values), I could have restored them individually if I wanted that ability.

Mover Controls

Several of the classes in the User Controls subfolder are “mover” classes, components that present the traditional two lists (one of available items and one of selected items) and buttons for moving items between the lists. “Mover” (the _Mover class) is the parent class for all of these classes, all of which are defined in _MOVERS.VCX. It has the following methods:

- InitChoices: sets the list of available items (the left listbox) to the values in the passed array. If the SortLeft property is .T., the list will be sorted. If the UseArrays property is .T. (the default), the passed array is copied to the aChoices array property and the listbox is based on this array; otherwise, the entries in the passed array are copied to the list using AddItem. If some items are already selected, they shouldn’t be in the array passed to this method.

- InitSelections: sets the list of already selected items (the right listbox) to the values in the passed array. As with InitChoices, the passed array is copied to the aSelections array property and the listbox is based on this array if the UseArrays property is .T.

- SizeToContainer: sizes and positions all the objects appropriately based on the container size.

- GetSelections: populates the passed array with the selected items and returns the number of items. The right list has mover bars, so the user can rearrange the list; GetSelections returns them in the order they appear in the list.

_Mover has label objects (Label1 and Label2) above the lists that by default have no Caption values set but can be used as heading for the lists. It also has an Updated property that contains .T. if the user added items to or removed items from the selected list.

There are a couple of bugs in _Mover. Unfortunately, you have to fix these bugs right in the class rather than creating a subclass and fixing them there because the other mover classes in _MOVERS.VCX are subclasses of _Mover. The PopList method, which populates a listbox with items in a passed array, doesn't handle singly dimensioned arrays correctly. Here's the fix:

```
LPARAMETER aListArray,oLstRef
EXTERNAL ARRAY aListArray

private cTmpListStr,i
m.cTmpListStr = ""

*** NEW CODE
if alen(aListArray, 2) = 0
    for m.i=1 to alen(aListArray)
        m.oLstRef.AddItem(aListArray[m.i])
    endfor
else
*** END OF NEW CODE
    for m.i=1 to alen(aListArray,1)
        m.oLstRef.AddItem(aListArray[m.i,1])
    endfor
*** NEXT LINE IS NEW
endif alen(aListArray, 2) = 0
```

The Click method of cmdRemove doesn't properly sort the left list when an item is moved from the right list to it. Here's a snippet of code from this method that shows the fix:

```
* IF this.parent.lstLeft.sorted
IF this.parent.SortLeft
    =ASORT(this.parent.aChoices)
ENDIF
```

You might also want to set the IntegralHeight property for the two listboxes to .T. while you're at it.

MOVER1.SCX, shown in Figure 1, shows how _Mover is used. It displays a list of available company names and those already selected (I just picked a few at random). The form's Init method has the following code:

```
local laChoices[1], ;
    laSelections[3]
use (_samples + 'DATA\CUSTOMER')
select COMPANY from CUSTOMER into array laChoices

* Preselect a few of these and remove them from the
* choices list.

laSelections[1] = laChoices[4]
laSelections[2] = laChoices[8]
laSelections[3] = laChoices[11]
adel(laChoices, 11)
adel(laChoices, 8)
adel(laChoices, 4)
dimension laChoices[alen(laChoices) - 3]

* Now populate the mover lists, set captions for the
* labels, and size it appropriately.

with This.oMover
    .InitChoices(@laChoices)
    .InitSelections(@laSelections)
    .Label1.Caption = 'Available companies:'
    .Label2.Caption = 'Selected companies:'
    .SizeToContainer()
endwith
```

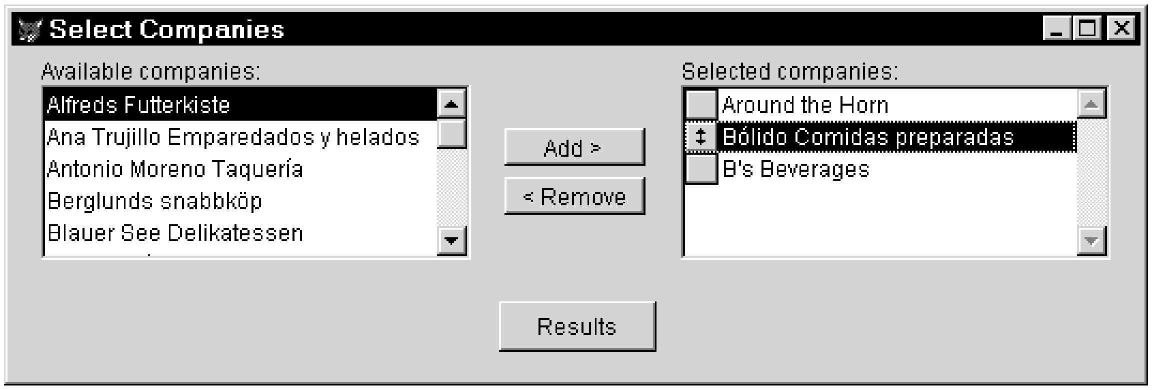

Figure 1. MOVER1.SCX demos the _Mover class.

Try moving some items between the lists and rearrange the items in the right list. Click on the Results button to display whether you made any changes or not, how many companies are selected, and what their names are.

“Super Mover” (the _SuperMover class) is a subclass of _Mover that has additional behavior: it has “add all” and “remove all” buttons (and all buttons have pictures rather than text prompts) and supports a maximum number of items that may be selected (the MaxItems property with the error message to display in the MaxMessage property). Like _Mover, it has a bug dealing with sorting items in the left list after they’ve been moved from the right. The following snippet shows the fix for the Click method of cmdRemoveAll:

```
*lSorted = this.Parent.lstLeft.Sorted
lSorted = this.Parent.SortLeft
```

To see how _SuperMover works, right-click on the class in the Component Gallery and run the sample from the context menu that appears.

“Field Mover” (the _FieldMover class) is a subclass of _SuperMover, “Table Mover” (_TableMover) is a subclass of _FieldMover, and “Sort Mover” (_SortMover) is a subclass of _Mover; these classes provide field mover, field mover with database and table selection, and field and index mover features. The main problem with these classes is that they use actual field names rather than their captions. As a result, your users have to know the actual names of fields rather than the nice names they probably know them by. Also, there's no facility for excluding certain fields, so the user will see primary key fields even if they never see them anywhere else in the application. These problems severely limit the usefulness of these classes in anything but tools intended for other developers (these classes are used in various wizards included with VFP), which is too bad, because I can see these classes could have been useful for things like allowing the user to select which fields should appear in a grid or how a report should be sorted.

Let's subclass _FieldMover to give us some new functionality: displaying field captions rather than field names and providing the ability to remove certain fields from the list. SFFieldMover (in SFFFC.VCX) overrides the GetTableData method, which reads the structure of a table and adds the fields to the list boxes. Here's a snippet from this method that has the changed code:

```
for lnI = 1 to fcount()
    lcField  = field(lnI)
    lcAField = alias() + '.' + lcField
    if This.IsFieldValid(lcAField)
        lcCaption = ''
        if not empty(dbc()) and indbc(lcAField, 'Field')
            lcCaption = dbgetprop(lcAField, 'Field', ;
                'Caption')
        endif not empty(dbc()) ...
        lcCaption = iif(empty(lcCaption), ;
            proper(strtran(lcField, '_', ' ')), lcCaption)
        lnFields = lnFields + 1
        dimension aPickFields[lnFields]
        dimension This.aFieldList[lnFields, 2]
        aPickFields[lnFields] = lcCaption
        This.aFieldList[lnFields, 1] = lcCaption
        This.aFieldList[lnFields, 2] = lcField
    endif This.IsFieldValid(lcAField)
next lnI
```

Since we need to keep track of both the field caption and name, I added a new aFieldList array property to the class. The code shown above gets the caption of the field from the database container (or uses a "cleaned up" field name if it's a free table or the caption is blank) and adds the caption to aPickFields (which will later be used to populated the available fields list box) and the caption and field name to This.aFieldList. I also added an abstract method, IsFieldValid, which determines if the field can be added to the list. This method simply returns .T. in this class, but in a subclass or instance, you can put code (typically in a CASE statement) that returns .F. for certain fields. This can be hard coded (by comparing the passed field name to a list of field names) or data-driven (using, for example, DBCX properties to determine if the field should be used; see the Technical Papers page of www.stonefield.com for an article on DBCX).

I also overrode the GetSelections method since we want to return an array of field names, not the captions. Here's the code for that method:

```
lparameters taFields
local lnCount, ;
    laFieldList[1], ;
    lnI, ;
    lcField, ;
    lnField
with This
    lnCount = .lstRight.ListCount
    if lnCount > 0
        acopy(.aFieldList, laFieldList)
        dimension taFields[lnCount]
        for lnI = 1 to lnCount
            lcField = .lstRight.List[lnI]
            lnField = .AColScan(@laFieldList, lcField, 1)
            taFields[lnI] = iif(lnField = 0, '', ;
                .aFieldList[asubscript(.aFieldList, lnField, ;
                1), 2])
        next lnI
    endif lnCount > 0
endwith
return lnCount
```

MOVER2.SCX, shown in Figure 2, shows how SFFieldMover is used; it displays fields from the CUSTOMER sample table that comes with VFP. The IsFieldValid method of the SFFieldMover instance prevents the CUST_ID and MAXORDAMT fields from being available. The code in the Init method of this sample form is very simple:

```
open database (_samples + 'DATA\TESTDATA')
use CUSTOMER
This.oFieldMover.InitData()
This.oFieldMover.SizeToContainer()
```

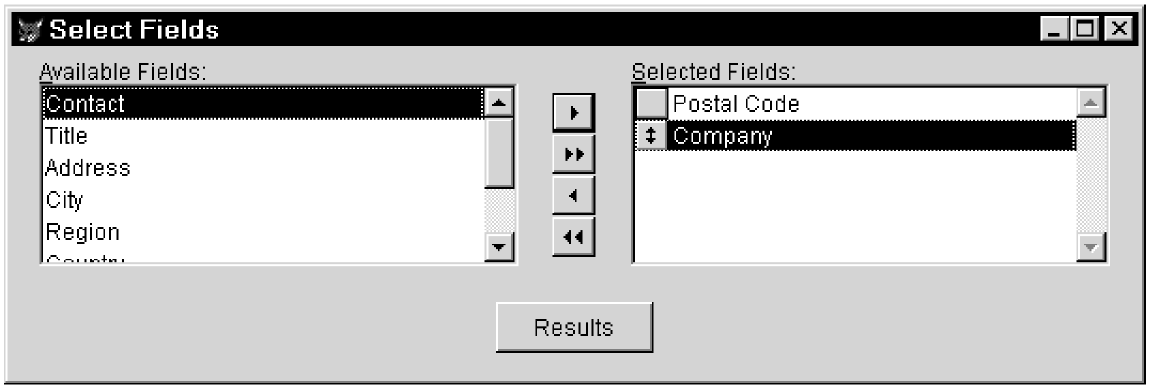

Figure 2. MOVER2.SCX demos the SFFieldMover class.

Text Formatting

The Text Formatting subfolder of the Component Gallery contains several controls, all of which are contained in _FORMAT.VCX, for formatting text. “Font Combobox” (class _cboFontName) contains a list of fonts installed on the system (using VFP’s AFONT() function) and “Fontsize Combobox” (_cboFontSize) contains font sizes for a specific font. “RTF Controls” (_RTFControls) and “Format Toolbar” (_tbrEditing) are both toolbars consisting of controls for text formatting, but _tbrEditing is the more feature-rich of the two. It has _cboFontName and _cboFontSize controls, command buttons for bold, italics, and underline settings, and a combobox for setting foreground and background colors. The toolbar has an nAppliesTo property that contains 1 if the settings apply to the current control in the active form, 2 if they apply to all textbox and editbox objects in the form, or 3 if they apply to all controls in the form. The InteractiveChange event of each control adjusts the appropriate property of the appropriate controls (based on nAppliesTo) in the active form. The _cboFontName object also adjusts the _cboFontSize object so it displays sizes for the selected font.

Although a sample is registered for these classes, it doesn't show how changing nAppliesTo affects the formatting, so I whipped up a quick and dirty demo, EDITING.SCX, shown in Figure 3, to do that. Type some text into the textbox and editbox, then use the toolbar to change the formatting. Choose a different option in the optiongroup and notice that formatting then applies to different controls (even the option buttons). Notice that the toolbar reflects the settings of the current control; this is done by refreshing the toolbar in the GotFocus method of each control.

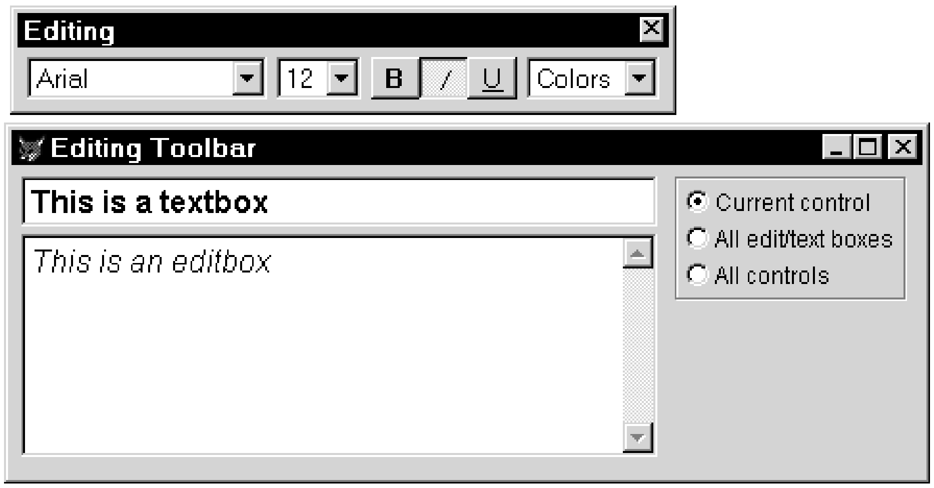

Figure 3. EDITING.SCX demos the _tbrEditing class.

I haven't used the toolbar for an actual project, but I have used the _cboFontName and _cboFontSize classes in a preferences form where the user could choose the font and size for grids in an application. These settings were stored in the Registry and restored in the Init method of the grid class.

Menus

The Menus subfolder has two classes: Shortcut Menu Class (_ShortcutMenu in _MENU.VCX) and Navigation Shortcut Menu (_NavMenu in _TABLE2.VCX). ShortcutMenu is an object-oriented wrapper for VFP's non-object-oriented shortcut menu system. It's one of the most useful classes in the FFC because it provides a simple way to programmatically (and therefore dynamically) create shortcut menus. The sample that demonstrates it is pretty good; you can run it from the Component Gallery by right-clicking on the class, choosing View Sample, and then Run. This class has the following methods:

AddMenuBar: adds a new bar to the menu. You can specify the prompt and optionally the code to execute when the bar is selected (if nothing is specified, the code in the cOnSelection property is executed) or another menu object to display as a submenu, other clauses for the bar (such as a SKIP FOR clause), the location of the bar in the menu, whether the bar is marked or not, whether the bar is disabled or not, and whether the bar appears in bold or not.

AddMenuSeparator: adds a separator bar to the menu.

ShowMenu: displays the menu, executes the appropriate action for the selected bar, and then hides the menu.

ClearMenu: removes all bars from the menu.

NewMenu: instantiates another _ShortcutMenu object without having to use CREATEOBJECT() or NEWOBJECT(). This is frequently used when a menu bar has a submenu. For example, here's code from the _NavMenu class that defines a submenu (oGoMenu) first, then the main menu (This.oMenu), with the submenu used as the action to execute when the first bar in the main menu is selected (goMenu is passed as the action in the AddMenuBar method).

```
oGoMenu = THIS.oMenu.NewMenu()
WITH oGoMenu
     .AddMenuBar(MENU_TOP_LOC,"oTHIS.oNav.GoTop()")
     .AddMenuBar(MENU_BOTTOM_LOC,"oTHIS.oNav.GoBottom()")
     .AddMenuBar(MENU_NEXT_LOC,"oTHIS.oNav.GoNext()")
     .AddMenuBar(MENU_PREV_LOC,"oTHIS.oNav.GoPrevious()")
     .AddMenuBar(MENU_RECORD_LOC,"oTHIS.DoGoto")
ENDWITH

WITH THIS.oMenu
     .AddMenuBar(MENU_GOTO_LOC,oGoMenu)
     .AddMenuSeparator
     .AddMenuBar(MENU_ADD_LOC,"oTHIS.AddRecord")
     .AddMenuBar(MENU_DELETE_LOC,"oTHIS.DeleteRecord")
     .AddMenuSeparator
     .AddMenuBar(MENU_SORT_LOC,"oTHIS.DoSort")
     .AddMenuBar(MENU_FILTER_LOC,"oTHIS.DoFilter")
     .AddMenuBar(MENU_FILTER2_LOC,"oTHIS.DoFilter2")
ENDWITH
```

_ShortcutMenu is pretty complete, except for a couple of things. First, you have to know the name of the array containing the menu choices (it's called aMenu) and do ALEN() on it to know how many bars have been defined. Wouldn't you already know how many bars there are since you populated the menu yourself? Not necessarily; we may want to populate the menu, then allow a hooked object to add its choices to the menu. So, I subclassed _ShortCutMenu into SFShortcutMenu (in SFMENU.VCX) and added an nBarCount property, with access and assign methods. The assign method has this single line of code:

```
error 1743, 'nBarCount'
```

This makes the property read-only. The access method has the following code:

```
return iif(empty(This.aMenu[1]) or ;
    isnull(This.aMenu[1]), 0, alen(This.aMenu, 1))
```

This returns the number of rows in the aMenu array if there are any bars defined and 0 if not. Notice that this code doesn't actually assign a value to nBarCount; it always remains 0, but the access method calculates and returns a value when anything asks for it, so it's not important that the property actually contains any value.

The second issue is that you can't change the settings of an existing bar without clearing the menu and refilling it. I added two methods to support this: FindBar, which looks in the aMenu array to find the specified prompt and returns the row number, and EditMenuBar, which allows you to change any of the settings for the specified bar number.

I like _ShortcutMenu (and SFShortcutMenu) so much that I implemented it in every class in SFCTRLS.VCX (my base class library). I added an oMenu property to contain an object reference to an SFShortcutMenu object and an lUseFormShortcutMenu property, which contains .T. if the menu of the form should be included in the menu for an object on that form. I made the Destroy method set oMenu to NULL to avoid dangling object references. I added a custom ShowMenu method (which is called from RightClick) that instantiates an SFShortcutMenu object, calls the custom ShortcutMenu method to populate it, calls the ShortcutMenu method of a hooked object (if there is one) to add to it, and then displays the menu if it has any bars. Here's the code:

```
private loObject, ;
    loHook, ;
    loForm
with This
    loObject = This
    loHook   = .oHook
    loForm   = Thisform

* Define the menu if it hasn't already been defined.

    if vartype(.oMenu) <> 'O'
        .oMenu = newobject('SFShortcutMenu', 'SFMENU.VCX')
    endif vartype(.oMenu) <> 'O'
    if vartype(.oMenu) = 'O'
        if.oMenu.nBarCount > 0

* Populate it using a custom method.

            .ShortcutMenu(.oMenu, 'loObject')

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

            if vartype(loHook) = 'O' and ;
                pemstatus(loHook, 'ShortcutMenu', 5)
                loHook.ShortcutMenu(.oMenu, 'loHook')
            endif vartype(loHook) = 'O' ...

* If desired, use the form's shortcut menu as well.

            if .lUseFormShortcutMenu and ;
                type('Thisform.Name') = 'C' and ;
                pemstatus(loForm, 'ShortcutMenu', 5)
                loForm.ShortcutMenu(.oMenu, 'loForm')
            endif .lUseFormShortcutMenu ...
        endif.oMenu.nBarCount > 0

* Activate the menu if necessary.

        if .oMenu.nBarCount > 0
            .oMenu.ShowMenu()
        endif .oMenu.nBarCount > 0
    endif vartype(.oMenu) = 'O'
endwith
```

Note the use of PRIVATE, rather than LOCAL, variables. They define references to objects in case the action for a bar is to call a method of the object. You can't do this using code like "This.Method()" because "This" isn't applicable in a menu. Instead, the object reference is put into a variable and that variable is referenced; for example, "loObject.Method()".

The ShortcutMenu method is used to actually fill the shortcut menu object with menu bars. It accepts two parameters: an object reference to the menu object to populate and the name of the variable containing the object reference to this object. Why pass the menu reference when it's stored in the oMenu property? Because we might want to hook several classes together, populating the menu object of the first one. For example, you might want to have the right-click menu for a textbox include not only items for the textbox, but for the form it's on as well. To do that, set the lUseFormShortcutMenu property of the textbox to .T. The ShowMenu method of the textbox will call the ShortcutMenu method of the form, passing a reference to the textbox's menu object and the name of the variable the textbox defined that references the form (loForm). That way, the form can add its items to the textbox's menu and everything will work.

In most classes, ShortcutMenu is an abstract method since I couldn't think of any useful menu choices that'd be used in every instance and subclass. Those that have a text component (SFComboBox, SFEditBox, SFSpinner, and SFTextBox), however, have menu bars for cut, copy, paste, clear, and select all. Here's the code from SFTextBox.ShortcutMenu:

```
lparameters toMenu, ;
    tcObject
with toMenu
    .AddMenuBar('Cu\<t', ;
        "sys(1500, '_MED_CUT', '_MEDIT')")
    .AddMenuBar('\<Copy', ;
        "sys(1500, '_MED_COPY', '_MEDIT')")
    .AddMenuBar('\<Paste', ;
        "sys(1500, '_MED_PASTE', '_MEDIT')")
    .AddMenuBar('Cle\<ar', ;
        "sys(1500, '_MED_CLEAR', '_MEDIT')")
    .AddMenuSeparator()
    .AddMenuBar('Se\<lect All', ;
        "sys(1500, '_MED_SLCTA', '_MEDIT')")
endwith
```

Other obvious choices for menu items are add, delete, save, cancel, and other methods for data entry form classes, “dial this number” for a phone number class, and “send email” for a email address class.

To see an example of this class in action, run the ResizeDemo form and right-click on the upper textbox (which has its own menu only), on the form, and on the lower textbox (which includes the form’s menu, since it’s lUseFormShortcutMenu property is .T.).

_NavClass, the other class in the Menus folder, is more of a demo of how to use the _TableNav FFC class, which provides table navigation methods (like GoTop, GoNext, GoPrevious, etc.) than a useful class on its own, mostly because it lacks the flexibility you’d likely need in a real-life application. However, it might be worth looking at to get ideas for how you might create your own, more flexible version.

Utilities

We’ve already looked at the Registry classes in the Utilities subfolder; let’s look at some of the other ones.

Array Handler (the _ArrayLib class in _UTILITY.VCX) has three array-handling methods, which are very useful if you don’t already have routines that perform these functions. AColScan looks for a value in a particular column of an array (as opposed to ASCAN, which will look in every element of the array) and can return the row number it was found in (as opposed to ASCAN, which returns the element number that you almost always have to then use ASUBSCRIPT on to get the row number). InsAItem inserts an element into an array and sets that element to the specified value; in the case of a 2-dimensional array, it adds an entire row and can set either the first element or each element in the array to the specified value. DelAItem deletes an element (an entire row in the case of a 2-dimensional array) from an array.

Find Files (_Filer in _UTILITY.VCX) is a wrapper class for FILER.DLL, a cool text search tool found in the TOOLS\FILER subdirectory of the VFP home directory. Although you can use FILER.DLL directly, _Filer provides an instantiation wrapper for this COM object: if it isn’t registered on the system, it offers to register it for you (assuming the DLL can be found, of course). It has properties that map to the COM object’s properties; for example, _Filer.cFileExpression maps to the FileExpression property of the COM object. It can automatically handle empty properties in some cases; for example, if cFileExpression is blank, it puts “*.*” into the FileExpression property of the COM object, and if cSearchPath is empty, it can optionally display a dialog asking for the directory to search. It has one method you’ll use, Find, which returns the number of files it found and put an object reference to the Files collection of the COM object into its oFiles property. One major drawback with this class (and FILER.DLL): Microsoft doesn’t allow us to distribute the DLL, so you can’t use it to provide text search capabilities in your applications; you can only use it to create tools for yourself or other VFP developers.

Shell Execute (_ShellExecute in _ENVIRON.VCX) is a great little class. It provides a wrapper for the ShellExecute Windows API function, which “executes” a file, so you don’t have to worry about how to DECLARE or call it. The neat thing about this function (and the class) is that it handles file types registered on your system; passing the file name to ShellExecute will run the associated application if the file isn’t an EXE. For example, executing SAMPLE.HTML will bring up the default browser for your system and display this file (see TESTSHELL.PRG for code to do this). You don’t have to know which application is required or where it’s installed on the system. _ShellExecute just has one method, ShellExecute, which accepts up to three parameters: the name of the file to execute, the working directory to set for the application (if it isn’t specified, the application’s directory is the working directory), and an “operation” string (“open” is used if it isn’t specified). This method returns a value indicating success or what type of error occurred (see the help topic for this class for a list of values).

Incidentally, _ENVIRON.VCX contains a couple of other classes, neither of which appear in the Component Gallery. _Set stores the current value for a SET setting (such as TALK, PRINT, NEAR, etc.) and sets it to another value. When the object is destroyed, it automatically restores the setting (although this behavior can be disabled if desired). Using this class saves having to use code like the following:

```
lcCurrExact = set('EXACT')
set exact off
* do something here
if lcCurrExact = 'ON'
    set exact on
endif lcCurrExact = 'ON'
```

This code would be much worse if you had multiple exit points for the routine; you’d have to reset the EXACT setting in each place (yet another reason to avoid multiple exits in a routine). Instead, you’d just do the following:

```
loExact = newobject('_Set', '_ENVIRON.VCX', '', ;
    'EXACT', 'OFF')
* do something here
```

When this code ends, loExact goes out of scope and EXACT is set back to its former setting.

_RunCode, the other class in _ENVIRON.VCX, is a code block interpreter. As you know, you can make VFP execute a single line of code at runtime by storing it into a variable and then macro expanding the variable; this allows you to determine what code should execute at runtime rather than design time. However, this technique doesn’t work if the code is several lines long (for example, some “script” code stored in a memo field in a table that should be executed when a record is selected). While you could split the code into individual lines and macro expand each one, this won’t work if the code contains program structures such as loops. The _RunCode class has a RunCode method that can execute a block of code. You pass the code as a string or the name of a text file containing the code as the first parameter, .T. for the second parameter if the first parameter is a file name, and .T. if you want errors to be ignored. VFP 6 Service Pack 3 added the ability to compile code in a runtime environment, so this class isn’t really required anymore, but can still be useful in versions prior to that release.

One use for _RunCode would be to allow users of your application to script certain behaviors (not a typical end-user, but perhaps an administrator or another developer who doesn't have access to the source code). To do this, you'd provide a hook or "add-in" mechanism for the functions the user can change the behavior of. For example, say you allow the user to define add-ins for the "add record" method of a form. That method might look like this:

```
if This.oAddinManager.Execute('AddRecord')
    return
else
    append blank
    This.Refresh()
endif This.oAddinManager.Execute('AddRecord')
```

In this scenario, the form has an add-in manager that, when passed a method name, checks an add-ins registry (it could be a table or the Windows Registry) to see if an add-in has been defined for this method, and if so, executes it and returns .T. so the calling routine knows the add-in executed. The add-in might be code written by the administrator that does something like this:

```
if oUser.nUserLevel = 1
    do form ADDWIZARD
    return .T.
else
    return .F.
endif oUser.nUserLevel = 1
```

In this case, entry-level users would get a wizard that leads them through what information is captured when a record is added; the normal "add" operation is used for everyone else. The add-in code might be stored in a file (in which case the add-in registry would contain the name of the file) or in a memo field of an add-ins table (which would require providing a means for the user to edit the contents of the memo).

Summary

The FFC contains a lot of useful classes. Some of them are great as is, while others can be subclassed to add minor improvements. Either way, I suggest you spend some time looking at these classes and thinking about how you might use them. I'm sure you'll find some gems in the pile.

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He writes the monthly "Reusable Tools" column for FoxTalk, and is the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP).