Christmas Stocking Stuffers

Doug Hennig

Visual FoxPro has a lot more places to put code than FoxPro 2.x. This month's column examines the advantages and disadvantages of creating classes for library routines. It also looks at a replacement for FoxPro's GO command and shows how to make it more obvious that Visual FoxPro controls are read-only.

This is my "Christmas stocking stuffers" column, a potpourri of idea tidbits rather than one specific topic (as I write this column, it's about 33C (91F) outside, but I'm trying to picture snow and mistletoe to put me in the Christmas mood). We'll look a bunch of things, such as where to put code, why you shouldn't use the GO command, and how to visually indicate that a control is read-only.

Intellidrop Redux

In last month's column, I discussed the new Intellidrop feature in VFP 5.0. This feature allows you to define what class to use when a field is dragged from the DataEnvironment, Project Manager, or Database Designer and dropped onto a form. At the time I wrote the column, I was using an early beta version of VFP 5, and was disappointed at a few shortcomings in Intellidrop, such as the fact that it names the control by appending an instance number to the class name (such a SFTextbox1) and that the label it creates is of the Label base class rather than a class you can define. I'm happy to report that both of these have been improved: Intellidrop now names the control by using the field name with one of the standard naming convention prefixes (such as "txt" for a Textbox-based class), and allows you to define the class to use for the label in the Field Mapping page of the Tools Options dialog. You'll love the way Intellidrop greatly improves your productivity in creating forms!

Where Should Code Go

VFP has made some design decisions more complicated than they were before. In FoxPro 2.x, the only decision you had to make about where to put a function was whether it should go into a screen snippet, a standalone PRG, or a procedure file. In VFP, we still have these choices (although a form or object method replaces a screen snippet, of course), but we have the additional choice of putting functions into the methods of classes in class libraries (VCX files). We also have to split this into two different considerations: database-oriented functions (such as triggers and rules) and everything else.

Stored Procedures or External Code

First consideration: where should generic routines called from triggers, field or table validations, or default expressions be stored: in the stored procedures of the database or somewhere external (such as a PRG file)? An example of a generic routine is a "next available primary key" routine that, given a table name passed as a parameter, returns the next primary key to use by looking it up in a file containing these values.

The advantage of putting such routines in the stored procedures of a database is the database becomes a self-contained entity; it doesn't rely on outside programs for anything. However, there’s a big drawback: every database using these generic routines has to duplicate the code within its stored procedures. If you discover a bug or wish to enhance a generic routine, you’ll have to find every database containing the routine and apply the same changes. What a maintenance headache!

The approach I use is to put routines enforcing business rules specific to a database in the stored procedures for the database, and put generic, reusable code where it belongs: in library routines. The drawback to this approach is the database can’t stand by itself; the library routines have to be available or the stored procedures calling them will fail. However, since I rarely allow users of applications I write to browse tables directly from the Command window, I don’t consider this to be a major concern. Also, if someone does browse a table and try to make changes, the trigger and validation routines will fail (an error trap can provide a more elegant interface than just the VFP error dialog) and no changes to the data will be permitted. In other words, if we can’t ensure the data is valid, let’s not allow it to be changed.

Function or Object

Second consideration: should you use the FoxPro 2.x method of creating function libraries in PRGs or should you create classes of library routines?

To create a “function library” class, you typically create a class that has Custom as its base class and add methods that provide the individual functions you need. The methods in each class should have something in common (for example, they all perform different date manipulations or low-level file functions). Different classes can be combined into one or more class libraries.

There are advantages and disadvantages to putting library routines into classes. Here are some advantages:

- Properties of a class act like static variables. For example, you might want to use a default title for a set of messaging functions. By defining these functions as methods in a Message class and creating a cDefaultTitle property used by each method, you don’t have to pass a title to a method unless you want to override the default title.
- Extending the first point, you can configure the behavior of an object by setting properties rather than having to pass a parameter to each function. For example, several methods in one of my classes decide whether to display a status message based on the value of an lQuiet property. If I created these methods as functions instead, each would have to accept an additional parameter to replace this property.
- You can subclass a library class to obtain new behavior. For example, you could subclass a class that prints spelled-out dollar amounts for checks if you need to handle different languages or currencies. As another example, I’ve written functions to which I had to add additional parameters every time I wanted to add new optional functionality. As the features list grew, the list of parameters grew to be almost unmanageable. If I’d created a class with the base functionality, I could create subclasses when more complex functionality was needed while still keeping the simpler class for simple uses.

- Steve Black's excellent series in FoxTalk on design patterns has shown us that you can create more flexible classes by making objects communicate to extend functionality than you can by subclassing.
- Here's a packaging issue: you could create MotherOfAllLibrary.VCX containing lots of utility classes that have lots of methods, but you can also have MotherOfAllLibrary.PRG which has all the same functions inside it. However, individual functions could be grouped by type (file utilities, string utilities, etc.) and combined into classes inside the VCX file. This is easier to manage than one huge PRG file. Also, you can easily move classes from one VCX to another if necessary; moving code from one PRG to another requires cutting and pasting between PRGs, including any subroutines related to the routine of interest.
- With a PRG library, you must SET PROCEDURE TO the PRG file before you can use any functions in it. With a class library, you can simply drop a "library" object on a form and don't even have to worry about SET CLASSLIB TO. Also, when should you release the procedure file? You can't necessarily do it in the Destroy method of a form, because another form that was opened after the current one may need the procedure file left open. This isn't an issue with objects dropped on forms (although it is if you use CREATEOBJECT to instantiate a class).
- You get additional automatic functionality through the Init and Destroy methods of a class. For example, a wrapper class for FOXTOOLS.FLL functions can automatically SET LIBRARY TO FOXTOOLS (including locating the FLL) in its Init method and automatically RELEASE LIBRARY FOXTOOLS in its Destroy method. Anything needing FOXTOOLS functionality can simply instantiate this class without having to worry about these setup and cleanup issues.

Here are some disadvantages of putting your functions into classes:

- Because FoxPro 2.x doesn't support VCX files, you have to maintain two different versions of code (a PRG for FoxPro 2.x and a VCX for VFP) if want to use a function in FoxPro 2.x.
- To use a function in the Command window, you have to SET CLASSLIB TO the VCX containing the class, instantiate the class, then call a method. With a PRG library, you just need to SET PROCEDURE TO the library program, then call the desired function (or just call the function directly if it's in its own PRG file).
- The syntax for an object method is more cumbersome than a function: Thisform.oLibrary.Function() instead of just Function().
- Object methods run slower than functions.
- Instantiated objects consume memory even when they're not being used, while functions only do so when executing.
- PRG files are smaller than VCX/VCT files (OK, with gigabytes drives costing about as much as a Snickers bar these days, maybe this isn't a big deal).

Although I've been wavering on this issue since I started using VFP, recently my preference has solidified to using classes rather than function libraries. It just seems a lot cleaner to me to drop a library object onto a form that needs the functions in the library than to worry about whether I've SET PROCEDURE TO the library PRG file. This isn't to say that I'll stop using PRGs. For example, my "next available primary key" routine is a standalone PRG because I can call it as the Default of a field without concern about whether an object has been instantiated or a function library opened.

Why You Shouldn't Use GO

If memory serves, the GO command has been in the Xbase language since day one. It simply moves the record pointer to the specified record number. However, if you try to GO to a record number that's invalid (zero or greater than the number of records in the table), or if no table is in use in the current work area, you'll get an error. While this may seem unlikely if you haven't run into it, consider a case like the following: you store the current record number into a variable, SEEK or LOCATE for a desired record, then GO back to the saved record number. What happens if the table was sitting at end-of-file when you saved the record number? What happens if there were no records in the table (the user was running the application for the first time)?

I've mostly stopped using GO and instead use a routine called REPOSN (short for "reposition"). It'll either move the record pointer to the desired record or to end-of-file if the record number is invalid. REPOSN accepts two parameters: the record number to move to (required) and the alias to move the record pointer for (optional: if it isn't specified, the record pointer in the current work area is moved). It works in both FoxPro 2.x and Visual FoxPro, and handles negative record numbers (which are used for inserted records when table buffering is turned on) in VFP. Here's the code for REPOSN (included on the source code disk):

```
parameters tnRecno, ;
  tcAlias
private lcAlias

* If no alias was passed, use the alias in the
* current workarea.

do case
  case type('tcAlias') <> 'C' or empty(tcAlias)
        lcAlias = alias()
  case not used(tcAlias)
        wait window 'Alias ' + tcAlias + ' not found.'
        lcAlias = ''
  otherwise
        lcAlias = tcAlias
endcase

* Move the record pointer.

do case
  case empty(lcAlias)
  case ('Visual' $ version() and tnRecno < 0) or ;
        between(tnRecno, 1, reccount(lcAlias))
        go tnRecno in (lcAlias)
  otherwise
        go bottom in (lcAlias)
        if not eof()
                skip in (lcAlias)
        endif not eof()
endcase
return
```

Creating Read-Only Controls

You may occasionally need to display read-only controls in a form. For example, if you have a TextBox where the user can type in the customer number for an order, you probably want to display the customer name in a read-only TextBox beside the customer number. There are three ways VFP allows you to create a read-only control, none of which are great:

- You can set the ReadOnly property of the control to .T. which prevents the user from editing the value and, in VFP5, makes the control have the same background color as the form (which makes it "look" read-only). Unfortunately, the control can still receive focus and it displays "The record is read-only" ("The control is read-only" in VFP 5) in the status bar if the user tries to edit the value. Also, this property is not available for all classes (such as CheckBoxes in both VFP 3 and 5, and ComboBoxes in VFP 3), in VFP 3 doesn't change the background color of the control, and still displays the default I-beam for the mouse pointer when it's over the control.
- You can set the Enabled property to .T., which prevents the control from receiving focus and makes the mouse pointer appear as an arrow when it's over the control. However, the default disabled colors are light grey text on either a white (VFP 3) or grey (VFP 5) background, which is difficult to read. Also, the control could become enabled again if, for example, something issued a Thisform.SetAll('Enabled', .T.) command.
- You can use RETURN .F. for the When method of the control. This prevents the control from receiving focus but the control doesn't visually appear to be disabled (the colors appear to be normal and the mouse pointer is still an I-beam), which will confuse the user.

What we *really* want is a control that appears with black text on a grey background (assuming the form has a grey background) because that's easier to read, can't receive focus, and displays an arrow for the mouse pointer. Because of differences between VFP 3 and 5, we have to use somewhat different approaches for each version. Here are the properties and methods for read-only versions of controls in VFP 3 (these would, of course, be created as classes rather than making changes to each control in a form):

Object	**Properties**	**Methods**

TextBox, EditBox	BackStyle = 0 (transparent)
	MousePointer = 1 (arrow)
		When: RETURN .F.

ComboBox, Spinner	MousePointer = 1 (arrow)	Init: This.BackColor = This.Parent.BackColor When: RETURN .F.

CheckBox, OptionButton	BackStyle = 1 (opaque)
	DisabledForeColor = 0,0,0
	Enabled = .F.
	ForeColor = 255,255,255
		Init: store This.Parent.BackColor to This.BackColor, This.DisabledBackColor
		When: RETURN .F.

OptionGroup	BackStyle = 0 (transparent)	When: RETURN .F.
ListBox		Init: This.ItemBackColor = This.Parent.BackColor When: RETURN .F.

Grid	MousePointer = 1 (arrow)	Init: This.BackColor = This.Parent.BackColor
Control in Grid column	MousePointer = 1 (arrow)	When: RETURN .F.

Here are the properties and methods for read-only controls in VFP 5:

Object	**Properties**	**Methods**

TextBox, EditBox, ComboBox, Spinner	ReadOnly = .T.
	MousePointer = 1 (arrow)
		When: RETURN .F.

CheckBox, OptionButton	BackStyle = 0 (transparent)
	DisabledBackColor = anything except BackColor value (eg. 255,0,0)
	DisabledForeColor = 0,0,0
	Enabled = .F.
		When: RETURN .F.

OptionGroup	BackStyle = 0 (transparent)	When: RETURN .F.
ListBox		Init: This.ItemBackColor = This.Parent.BackColor When: RETURN .F.
Grid	MousePointer = 1 (arrow)	Init: This.BackColor = This.Parent.BackColor
Control in Grid column	MousePointer = 1 (arrow)	When: RETURN .F.

CheckBoxes and OptionButtons are more complicated than other controls because they have two parts: a graphical part (the box or circle) and a text part (the caption). We want the caption to appear in black text but the graphical part to appear disabled. In order to do this, the control must be disabled but the color properties changed so the text appears “enabled”. This is especially complex because the settings of the various color properties interact with each other when the control is disabled. I’m not completely happy with the way these controls appear (in VFP 5, the caption has a white “shadow”) but that’s the closest I could get.

The source code disk includes a class library (CONTROLS.VCX) containing classes with these properties and methods set and a form (TESTRO.SCX) showing how various versions of the controls look and behave. There are separate VFP 3 and 5 versions of both of these.

If you want to create controls that can be dynamically made read-only (for example, if the user’s security level is too low, you want all controls on a form to be read-only), these classes will have to be changed somewhat. For example, for a dynamically read-only TextBox class, you would change the When method to RETURN NOT This.ReadOnly instead of RETURN .T. so the control can receive focus when it isn’t read-only. You’ll also need to change the MousePointer back to 0 (default) when the control isn’t read-only. This would best be handled in a custom method of the control (such as SetReadOnly).

Conclusion

Next month, we’ll take an in-depth look at “best practices” for working with arrays, including a library routine that does a better job of searching an array than ASCAN. In the meantime, I hope Santa is good to you!

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 Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. 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.