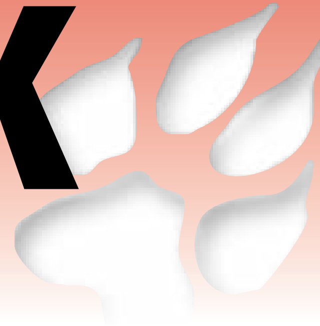
PINNACLE

FoxTalk

Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers



Now You See It, Now You Don't

Paul Maskens and Andy Kramek

Combo boxes provide the basic means for offering end users a list of valid options, but they can be slow and unwieldy to use with large numbers of choices. Grids are much better in these situations but often take up too much room on the form. Why not combine the two? …

Andy: Hi, Paul. I'd like to pick up on something that we discussed a while ago—your Report Selection combo boxes, which used hidden grids to handle a large number of options. I've been thinking about ways to improve the method of giving my users lists of available items when entering data.

Paul: Ah, yes! The basic idea came from Knowledge Base article Q155013, which described the methodology for creating a control that mimics the action of a combo box but uses a grid for the drop-down display portion.

Andy: Yeah, I read that article, and I tried it, but it didn't work "out of the box"—I forget why, and I never went back to it.

Paul: It appeared to have been mangled by the conversion of KB articles to the Web. If you already had a good idea what you were doing, it was possible to fix it—but then I suppose you didn't need the article <g>.

Although it worked, it was limited to searching in Column1. It didn't behave quite like a combo box either; I think I had to add the support for BackSpace to my version.

Andy: Ah! I see. Well, what I'm looking for is something that will allow users to choose from a list (which might be several thousand rows!). More importantly, I need to add more than one "column" of data when a new entry

Continues on page 4

July 1999

Volume 11, Number 7

1 **Now You See It, Now You Don't** *Paul Maskens and Andy Kramek*

3 **Editorial: Code Rage** *Whil Hentzen*

8 **Best Practices: Seeing Patterns: The Adapter** *Jefferey A. Donnici*

14 **Reusable Tools: Splitting Up is Hard to Do** *Doug Hennig*

19 **The Kit Box: I Was Framed!** *Paul Maskens and Andy Kramek*

23 **July Subscriber Downloads**

EA **Hacking the SQL Server Upsizing Wizard** *Jim Falino*

EA **VB for Dataheads: Real-World Usage of ADO in VB** *Whil Hentzen*

6.0 Applies to VFP v6.0 (Tahoe)
5.0 Applies to VFP v5.0
3.0 Applies to VFP v3.0
2.x Applies to FoxPro v2.x

DOWNLOAD Accompanying files available online at http://www.pinpub.com/foxtalk

UNIX MAC DOS WIN Applies specifically to one of these platforms.

Code Rage

Whil Hentzen

THERE'S a TV set in Graceland (Elvis Presley's old digs) that bears a very significant feature—a bullet hole. Evidently, Elvis saw something on the tube that he didn't like.

Do you ever get mad at the tube? I mean the computer tube, of course.

I've been seeing more and more evidence of this—people posting messages on electronic forums or newsgroups without thinking through what they wanted to say, or even people e-mailing each other, pressing Send before they should have. Drew Speedie referred to it as "Code Rage" at FoxTeach a couple of months ago.

It's pretty tough not to take it personally. You're just up on one of the newsgroups, trying to help out, and someone comes out of left field and flames you badly enough to make a sailor blush. Jeepers. Time to quit the volunteer help game, eh?

Before you get all worked up about being on the receiving end, step back for a second. Yeah, it's possible that the sender is a rotten person. There are a few of those out there—the ones who seem bent on taking something the wrong way, or venting their anger at anyone foolish enough to show up in public. But more likely, the person was just having a "bad code day." And you've had those as well, haven't you?

So instead of responding to a "nastygram" in the same vein, try helping them out of their funk. They've already lost the respect of many people for mouthing off inappropriately in public—why dig that hole deeper? Twenty-four hours later, they'll be really embarrassed and will appreciate your trying to help out, instead of returning the nastiness. If they continue to be nasty, you

New Pinnacle Newsletter, *ActiveWeb Developer*, Released

If you're like most developers, you probably work with many tools besides Visual FoxPro. As business turns to creating more applications for Internet/intranet use, you're probably gaining more experience with the tools used in that development environment.

Internet development is an exciting area to work in, not the least because the tools are still developing and there are so many neat toys to work with. Pinnacle is launching a new publication called *ActiveWeb Developer* to support Web application developers. The editor of the new publication is Bill Hatfield, author of *Active Server Pages for Dummies* and *Visual InterDev for Dummies* and editor of Pinnacle's own *Delphi Developer* newsletter.

Bill's mandate is to make *ActiveWeb Developer* "a conduit for developers who want to share their knowledge and experience with others in the Web development community." Bill's focus is on the professional developer, concentrating on intermediate to advanced topics with real-world hints and tips on creating more powerful, user-friendly Web applications. Topics that Bill is planning include Using COM Components with Microsoft Transaction Server (MTS), Really Cool Scripting Tricks, The Best Techniques for Database Access, and Java and COM: Portable Technologies?. Check out the complimentary issue of *ActiveWeb Developer* that's enclosed, or surf over to Pinnacle's Web site at www.pinpub.com. I don't think you'll be disappointed. ▲

Visual Studio SP 3 Released

Microsoft has just released Visual Studio SP 3. You'll probably want to grab this one, since there are a number of fixes that are specifically for Visual FoxPro. You can download it (68M minimum) or order it on CD. Start by heading over to http://msdn.microsoft.com/vfoxpro and following the appropriate links (which undoubtedly have changed at least twice between the time I wrote this and the time you received it).

SP 3 includes Service Pack 1 and 2 (you don't have to install those and then SP 3), and it covers all Visual Studio apps, including VFP, VB, and InterDev. There are also updated versions of Data Access Objects, HTML Help, Data Access Components, Scripting, and OLE Automation.

As always, back up the data on your system before installing the new Service Pack. ▲

can pigeonhole them into the “huge chip on their shoulder so I don’t want to have anything to do with this person again” slot and forget about them.

But what about those senders? If you’re potentially on the sending end, then you might consider a trick that Woodrow Wilson used when he was President of the United States way back when. He would write a venomous letter to an individual whom he felt deserved it. And remember, back in those days, one didn’t use a string of four-letter words to express anger—one had to be considerably more clever. So upon reflection, consider what a nasty letter that didn’t include profanity might have looked like—and how it would have been received in those times.

The trick Wilson used, however, was never to send them. He’d write the letter and put it in a desk drawer. And there it would stay—for good. Unfortunately, about 50 years after he died, someone found that stack of letters, and published a book containing the juiciest ones. Engaging reading, if you’re interested . . .

So go ahead and vent. But before you press Send, think about whether or not your mother would be proud of you for sending that missive. Then save that document somewhere other than Drafts, and call it a day. ▲

Now You See It . . .

Continued from page 1

is made in the list. Using a standard combo, you can only add data to *one* column in the lookup table unless you bring up an entry form when a new item needs to be added. So I’m going to have to make the drop-down portion a grid.

Paul: True, and you’re going to need a proper incremental search method, too—but wait, aren’t we getting ahead of ourselves here? We haven’t actually decided exactly what you want yet.

Andy: Okay, here’s my specification for a generic class:

- Incremental search in a text box with display of available items from an unrelated table;
- Add items to lookup (which might require multiple fields—for instance, Street, City, County, and Postal Code); and
- Take up no more room on the form than a single text box.

Paul: Sounds like we need to apply the “must, could, should” rules here (remember, “must do” responsibilities go into the class, “could do” things indicate a requirement for a subclass, and “should be done” items aren’t the responsibility of the class at all). We *must* have a text box into which we can enter data, a mechanism to search the lookup table, a grid to display the matching data, and the mechanisms to display and hide the grid.

Andy: Also, we *must* be able to add to the lookup data (and probably edit the existing data—though I’m not sure that’s a *must* do).

Paul: Well, if you’re going to allow adds, you’d better also allow an edit if only so that errors can be corrected <bg>. I think the way to handle this is to use a container that can have within it the Search text box, a grid to handle the display of available items, and a button to allow the user to open the grid. It should look like **Figure 1** when the grid is closed and like **Figure 2** when the grid is opened.

Andy: That’s exactly what I want! So how do we go about it? I suppose we’re going to use the container to be the basic controlling object here?

Paul: Indeed we will use the container. Though I think the controlling object is actually the user <g>. The container is going to have to do a number of jobs. First, it must set up the grid (invisibly, of course), and it must also display and hide the grid when required. For the incremental search,

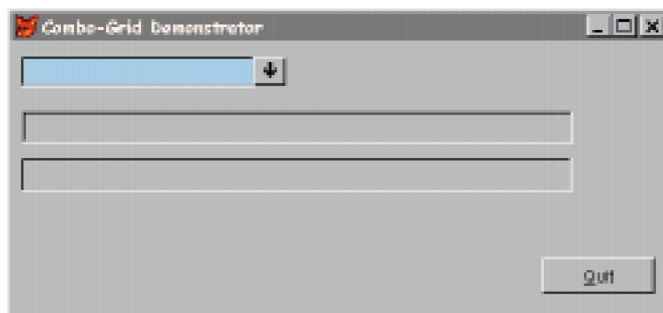


Figure 1. ComboGrid demonstrator form.

Figure 2. Dropped down ComboGrid.

you can use a "QuickFill" text box.

Andy: Okay, that'll work. So I need a method to initialize things, methods to open and close the grid, and a method to position the record pointer correctly in the grid's record source. Anything else?

Paul: How are you going to give the control all of the information it needs? You might (no, you *will*) want to add properties to the container and have the container's Init() populate the contained controls.

This container is going to function as if it were a single control—despite the fact that it's made up of a text box, a button, and a grid inside that container. Also, it's masquerading as a drop-down combo box, so that's a good reason to make it behave like one, with public PEMs to control its behavior.

Of course, I'm leaving aside the theoretical consideration of encapsulation.

So why not use the VFP control class? If you were using a control class, you'd have no access to the PEMs of the contained controls at all, even while debugging. That's why almost everyone uses the container class instead. In many ways, control is the right class to use.

Andy: But if you don't have access to the internal controls while you're debugging it, how on earth do you do it?

Paul: Ah—I cheat. I actually use a container while I'm developing, then when I have it all finished and tested, I simply change the class of the container (remember your HackVCX? I use it!) to be based on a control class. That doesn't change anything except the ability to access the controls individually, so it's actually no problem.

Andy: Hmmm. I'll have to think about that one—sounds a bit dicey to me! For the meantime, let's stick with the container as the outermost object, okay?

Now we'll need to indicate which column of the grid is going to be the one that returns the value, or were you thinking it would always be Column1?

Paul: No, let's be flexible here. You might want to display something like names in FirstName, LastName order and want the LastName returned. Limiting it to Column1 isn't good enough—after all, a standard combo box has a BoundColumn property to indicate which column to return as the Value.

I personally want to store the primary key in one of the columns and return that when selecting a value in the grid, so that ought to be an option, too.

Andy: Fine, we can use the properties on the container to control that.

Paul: Another point—when I implemented a class like this, I used a special container subclass that I called cntDroppable. I have to admit that I wasn't interested in the generic solution. (That's proper OOP for you, I suppose.) But I wanted to get away from setting design time properties where possible, so only the width was set at design time. The height was set by negotiating with the outer cntDroppable through a method to request the height needed for the control. That way, the ComboGrid could pop up if there wasn't enough room to drop down, because the cntDroppable returned control information saying how it got the requested size.

I don't want to add that sophistication to this example, but bear it in mind for the next version.

Andy: Hang on there, why do we suddenly have two containers? More to the point, why was the cntDroppable not generically reusable?

Paul: Well, it was a slightly strange requirement—but when have my problems been simple? My UI was designed around one-line containers, each of which could expand to show more lines of detail but by default showed the controls for the most common options. I used a "More >" button to show the extra controls for special purposes; of course, a "< Less" button collapsed the container. Somehow the name cmdMoreOrLess summed it up at the time. The outer container housed both the one-line display controls and the expanded controls, which were in their own expanding container.

However, I actually did end up reusing cntDroppable in several situations, not just for the ComboGrid, so I suppose it was fairly reusable.

Andy: Yes, I can see that a one-line display that expands to many is a slightly non-standard interface. An interesting digression, but a digression nonetheless. We're looking for a single control here.

Paul: Sorry, I thought you started it <g>. I wanted the ComboGrid to respect the size of the available space for the drop-down, without having to set it at design time. Back to our scheduled program . . .

Andy: Right, then—do you have any more requests?

Paul: I wanted to add multi-selection in a way that was actually usable, not the arcane Ctrl-Shift-Click MS method or the almost totally unusable keyboard equivalents. Let's not forget the keyboard user!

I added a column to the grid that contained check boxes that showed the selected status of a record and tied that into the keypress by subclassing the incremental search method of the container.

Yes, I know multi-select isn't a combo feature, but please don't do anything to make it harder for me to add it to a subclass, okay?

Andy: You don't want much, do you? Actually, I think your multi-select is really a function of a list box, not a combo box, and it ought to be a separate class. What you've mentioned are only "could" items, not "must" or "should."

Paul: Right, but would you accept them as valid subclasses? Extending the functionality of the ComboGrid into a DropDownListGrid? Having hooks on drop-down and popup so that behavior can be added to make the drop-down pop up instead? Having a hook in the keypress handling so {SpaceBar} can select or deselect an item?

Andy: I confess that I don't really agree here, but if you must, then you will anyway <g>. However, the objective of the ComboGrid is to allow the user to type something into the text portion and then carry out an incremental search in the grid and return the single row that matches, or to drop the grid open if no match is found. In other words, the idea here is to find *one* row, and one row only. Why would you ever want a multiple selection?

Paul: Well, MS *appears* to implement the drop-down list (now called ComboList in VFP 6 Help, I see) in the combo box class because it's controlled by the style property. I admit that's not a good reason to do anything, but using the combo as the parent for the drop-down list seemed valid. Perhaps they should both be subclasses of a more abstract parent?

Andy: But the purpose of the drop-down list is still to select a single item. The VFP ComboBox has an IncrementalSearch capability, but no MultiSelect! The only difference between the styles is whether the text box portion of the combo is enabled for user input. So in this sense, the combo has less functionality than the standard list box.

Paul: Ah, I suppose you're right, but you still haven't answered my question about whether both a combo grid (single row selection) and a combo list (multiple selection) could be considered valid subclasses of the same abstract class?

Andy: Oh, I see what you're getting at. Sorry! Yes, I suppose they could be, and if you think that this is something that you might need, then they probably should be. (Darn it! I wish I'd thought of that before I wrote all of the code.) I wish I'd thought of it like that before going down the specific class route. Ah, well—once again, it shows how important it is to do what I say and not what I do <g>.

Paul: Okay—so let's see what you've got, then.

Andy: Well, there's a fair bit of code here, so I'll just hit the high points. By the way, before you ask, yes, I did have some help with the Grid—Marcia Akins devised most of the code that deals with the handling of the grid portion, including the Add and Edit functionality.

Paul: No surprises there—after all, she's known as the "Queen o' the Grids" on the CompuServe Visual FoxPro Forum.

Andy: Okay, then, here we go. The container has a number of custom properties and methods, which are shown in **Table 1** and **Table 2** (see page 7).

Table 1. ComboGrid custom properties.

Property	Description
cAlias	Table name for the grid's RecordSource.
nColCount	Number of columns to include in the grid.
cColSource	Comma-separated list of fields for the grid columns.
llsDropped	Flag: set when the grid is visible, cleared when it's hidden.
cSchStr	Incremental search key (used by the QuickFill text box).
lSelected	Flag: cleared when the grid is dropped, set when the user makes a selection.
cKeyField	Name of the field whose value is used when a row is selected from the grid.
lAddNewEntry	If true, items that are typed into the Search text box and that aren't in the grid's RecordSource are added. Otherwise, the text box remains bound, but the new item isn't added to the grid's table.
cTagName	Initial sort order for the grid—used in DoSetup only.
cControlSource	ControlSource for the Search text box when it's to be bound (optional).
nCurrentRow	Save the current active row when selecting edit from the shortcut menu so we can cancel editing if the user switches to a different row.
lEditing	Flag: set if we're editing the current grid row.

Andy: So to set this up, all you do is set the RecordSource for the grid (*cAlias*), specify the number of columns (*nColCount*), the ControlSource for each in a comma-separated list (*cColSource*), the name of the field that's the bound column (*cKeyField*), and the initial sort order (*cTagName*). If you want to bind the Search text box to a separate table field, you can do so by setting the *cControlSource* property.

Paul: That's not too bad. How do you handle the setting of the height and width of the Grid?

Andy: Ah, that's all done in the *DoSetUp()* method. However, I cheat—I've pre-defined the height of the grid and only actually calculate the width:

Table 2. ComboGrid custom methods.

Method	Description
DoSetUp	Initialize controls.
DoDrop	Open the Grid control and make it visible.
DoPop	Close and hide the Grid control.
DoSearch	Search the grid data for specified characters.
GetItem	Return the nth item from a comma-separated list.
OnNavigate	Called from the grid's AfterRowColChange to update the Search text box.
DoAddNew	Add a new entry to the grid's RecordSource if THIS.lAddNewEntry is true.
RefreshControls	Refresh any controls on the form that need to be refreshed after the record in the grid's RecordSource changes. This is a Template method in the class; any behavior here will be instance-specific. Called by DoAddnew(), the grid's AfterRowColChange(), and the KeyPress() of the Search text box.
ProcessShortcutMenu	Called from the grid text box; right-click to display the shortcut menu and allow the user to edit the current grid row or re-order the grid if the current column has an index tag.
EditCurrentRow	Allow editing of the current grid row after selecting the Edit option from the shortcut menu.
SetOrder	Set new order for the grid based on the choice from the shortcut menu and the tag on the current grid column.
DoUpdate	TableUpdate the grid's RecordSource after editing a given row.

```
*** DOSETUP METHOD
LOCAL lcAlias, lcField, lnFieldLen, lnFieldWidth
LOCAL lcTagName, loObject, loCol, lnCnt
WITH This
   IF TYPE("THIS.grdItms") # "U"
     loObject = THIS.grditms
   ELSE
     RETURN .F.
   ENDIF
   *** Set container height first
   .Height = 190
   *** Set up the Grid
   lcAlias = .cAlias
   lcTagName = .cTagName
   loObject.RecordSource = lcAlias
   IF !EMPTY(lcTagName)
     loObject.ChildOrder = lcTagName
   ENDIF
   loObject.ColumnCount = .nColCount
   lnTotWid = 0
   *** Get width, based on width of field and
   *** font of control
   FOR lnCnt = 1 TO loObject.ColumnCount
      *** Grab a reference to the column
      loCol = loObject.Columns[lnCnt]
      *** Replace the default text box with our
      *** own class
      loCol.RemoveObject( loCol.Controls[2].Name )
      loCol.AddObject( "txtCol", "txtCboGrid" )
      *** Get the field name to use as a ControlSource
      lcField = .GetItem( .cColSource, lnCnt )
      *** Now calculate the width and set the
      *** properties for the column
      IF ! ISNULL( lcField )
         lnFieldLen = FSIZE(lcField, lcAlias) + 2
         IF INLIST( TYPE( lcField ), "D", "T" )
            lnFieldLen = lnFieldLen + 4
         ENDIF
         lnFieldWidth = (lnFieldLen * ;
              FONTMETRIC( 6, loObject.FontName, ;
              loObject.FontSize, "N" )) + 4
         loCol.ControlSource = lcAlias + "." + lcField
         loCol.Width = lnFieldWidth
         loCol.ReadOnly = .T.
         *** Keep track of the total width as we go
         lnTotWid = lnTotWid + lnFieldWidth
      ENDIF
   NEXT
   *** Now set the grid width
   .Width = lnTotWid + 26
   loObject.Width = lnTotWid + 24
   *** If txtSch is bound, must
   *** synchronize Grid's RecordSource
   SELECT (lcAlias)
   IF ! EMPTY(.cControlSource)
      .txtSch.ControlSource = .cControlSource
      LOCATE FOR UPPER(ALLTRIM(EVAL(.cKeyField))) ;
        == UPPER(ALLTRIM(.txtSch.Value))
   ENDIF
ENDWITH
```

Paul: Pretty good; I suppose we could easily add a height property if we needed it. But one thing strikes me here. What's to stop you from defining a list of columns so wide that the grid ends up wider than the form on which it sits?

Andy: Nothing at all—I just never thought of it. I guess we need to add some code to test for the width of the form and adjust the form width to accommodate it.

Paul: Oh, *no!* That would be awful! Why not just check the width of the grid and add a horizontal scrollbar if the total width exceeds the width of the form? I know scrolling grids aren't terribly user-friendly, but it's better than having your forms suddenly resizing themselves.

Andy: I suppose you're right. But you'd also have to add some height to the grid to allow for the scrollbar on the bottom. Here's the code for opening the grid and resizing the parent container:

```
*** DODROP METHOD*** Open the grid and make it visible
WITH This
   *** Don't drop if no data!
   IF RECCOUNT( .cAlias ) < 1
      WAIT WINDOW "No Lookup Data Found" NOWAIT
      RETURN
   ENDIF
   *** If in a resizable container, Resize
   *** the Container first
   IF LOWER( .Parent.ParentClass ) = "xcntstdautosize"
      .Parent.DoGrow( .Height + 10, .Width )
      *** Make background opaque
      .Backstyle = 1
   ENDIF
   *** Bring to front
   .Zorder(0)
   *** Open grid and associated label
   .lblPrompt.Visible = .T.
   .grdItms.Visible = .T.
   .grdItms.SetFocus()
   *** Set flags and button caption
   .lIsDropped = .T.
   .lSelected = .F.
   .cmdDrop.Caption = CHR(233)
   .cSchStr = RTRIM(.txtSch.Value)
ENDWITH
```

Continues on page 18

Seeing Patterns: The Adapter

Jefferey A. Donnici

Have you ever had an object in your application that became obsolete? How did it affect the rest of the application when it came time to replace the old object with something new? In this month's column, the "Seeing Patterns" series continues with a look at the Adapter pattern. This pattern addresses the problem of changing objects and interfaces in an application, thereby providing reduced maintenance costs as an application is changed and updated.

IT'S a rare application that doesn't get any sort of updates or enhancements after its initial development cycle. New features are added, existing features are enhanced, and bugs are fixed. But when an object in your system is replaced with a new or different one, how do you deal with that change in the rest of your code that collaborates with that now-missing object?

This month, the Best Practices column continues the "Seeing Patterns" series with the Adapter design pattern. This pattern addresses the problem of changing interfaces, as described in the previous paragraph. For those just joining this in-progress series, the intent is to discuss various object-oriented design patterns with Visual FoxPro examples. You might also find it helpful to look up the earlier Best Practices columns in this series, as they contain an introduction to object-oriented design patterns. All of the patterns covered in this series, including the Adapter pattern, were introduced in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (Addison-Wesley, ISBN 0-201-63361-2). Having a copy of the book, which is also now available through the publisher on CD-ROM, could prove helpful when reading the columns in this series.

In this series, I'm obviously using VFP as the development tool of choice for illustrating these patterns, but I should point out once again that all of these patterns are just as applicable in other object-oriented languages. Because "design pattern" is a term that generically refers to an object-oriented design "description," the examples used in this series aren't the only possible implementations of a design based on these patterns.

Structural patterns

The Adapter is one of several "structural" patterns described in the *Design Patterns* book, though it's not the first such pattern to be covered in this series. The first pattern introduced in the "Seeing Patterns" series was the Bridge pattern, covered in the November 1998 issue of *FoxTalk*.

The common theme that occurs with all structural patterns is that they deal specifically with how multiple objects collaborate to yield a larger structure. In the case of the Bridge, we saw how an object's abstraction was separated (into another object) from its interface so that one could change without affecting the other. As a whole, however, they form a structure that performs a single role within the system.

In *Design Patterns*, a distinction is made between structural *class* patterns and structural *object* patterns. With structural class patterns, inheritance is used so that a design-time mechanism is used to provide a larger structure without making a single class too large or unwieldy. Structural object patterns, however, deal with a runtime approach wherein multiple objects are combined to create new behaviors—behaviors that would cause a single class to be overly complex or lack cohesion. Where possible, a design based on object composition will typically be more flexible and provide greater reusability than a design based primarily on inheritance. As explained in *Design Patterns*, "The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition."

For reasons that will be clear later in this article, the Adapter is an example of a structural class pattern. The Bridge pattern, on the other hand, is a structural object pattern because it defines a runtime relationship between the behavior's abstraction (interface) and the implementation. Either of the two can be interchanged with other objects without the other object in the composite being aware of it, and, most importantly, this dynamic substitution can occur at runtime.

The Adapter

As its name implies, the purpose of the Adapter pattern is to "adapt" a class's programmatic interface to a different interface, so that the clients using the "adapted" class see the interface they expect. This allows classes that might

not otherwise be able to be used together to work with one another via the "adapter" class that provides a "translation" between the two (see **Figure 1**).

Another common name for the Adapter class in this pattern is "wrapper," which does a good job of describing the role that the class plays in a design based on this pattern. The class with the "incompatible" interface has another class "wrapped" around it so that Client classes can interact with it in a consistent and compatible way.

Depending on how great the differences are between the Adapter interface the client uses and the Adaptee interface that ultimately receives requests, the amount of "translation" done by the Adapter can vary. For example, if the only differences between the expected interface and the Adaptee's interface are the names of the public methods, then not much translation needs to occur. The role of the Adapter is to simply change the names of the methods and return values from the Adaptee back to the client.

If, on the other hand, the Adapter must change the data types of parameters passed by the client, or perhaps the data types of return values from the Adaptee, then the amount of "adaptation" performed by the Adapter is increased. Furthermore, the Adapter might provide a whole new set of functionality that isn't supported explicitly in the Adaptee. For example, the client might expect that a single method performs a certain operation, but the Adaptee doesn't have a similar method on its interface. In that case, the Adapter may call multiple Adaptee methods, perhaps even performing some intermediate calculations on its own, so that the interface the client expects is supported—albeit with a fair amount of help from the Adapter.

Additionally, the Adapter can actually perform "translation" services between the client class and *more than one* Adaptee class. In other words, the Adapter may serve as a Mediator for several classes so that a single interface is exposed to the client with multiple class definitions providing the expected functionality through the Adapter interface and back to the client. For more information on the Mediator design pattern, see the December 1998 Best Practices column, "Seeing Patterns: The Mediator."

Adapting and bridging

While the Adapter is a structural *class* pattern and the Bridge is a structural *object* pattern, they do have a few similar features. For starters, both are structural patterns, so they necessarily deal with combining multiple objects into a single composite structure providing specific functionality to the system. Additionally, both patterns provide a layer of abstraction between an object or composite and the client object(s) that use it. They also both focus primarily on moving messages from one object

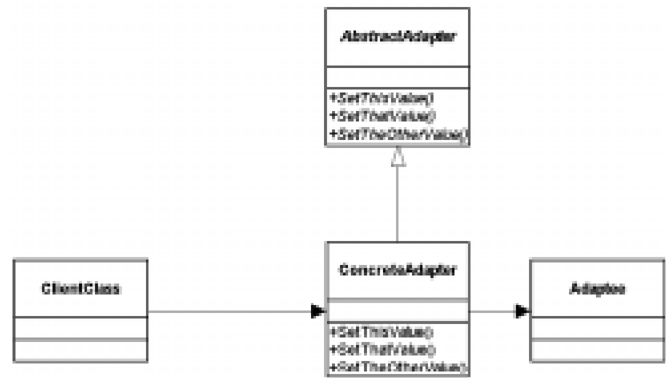


Figure 1. The ConcreteAdapter class provides translation services between the class being adapted (the Adaptee) and the Client class. When a new Adaptee is required, a different subclass of AbstractAdapter is used without any changes being made to the Client class.

to another, passing requests from the client object to the underlying objects that can act on that request.

The biggest difference between the two patterns is the purpose for which they're used. To put it simply, an Adapter pattern solves incompatibility problems between the programmatic interfaces of two classes, while a Bridge pattern lets the interface to a mechanism remain consistent to the clients while the implementation(s) can change dynamically. The following text from *Design Patterns* explains how the two are typically used to address widely different problems within an application: "[The] Adapter and Bridge are often used at different points in the software lifecycle. An Adapter often becomes necessary when you discover that two incompatible classes should work together, generally to avoid replicating code. The coupling is unforeseen. In contrast, the user of a Bridge understands upfront that an abstraction must have several implementations, and both may evolve independently. The Adapter pattern makes things work after they're designed; the Bridge makes them work before they are. That doesn't mean Adapter is somehow inferior to Bridge; each pattern merely addresses a different problem."

When and where

Much of the description in the *Design Patterns* discussion of the Adapter focuses on interfaces that simply "don't match." For example, when discussing the motivation for using the Adapter, the authors refer to a framework style class that "isn't reusable only because its interface doesn't match the domain-specific interface an application requires." This demonstrates the point that an Adapter is quite often used *after* an application is already in production. For example, the application needs to use a new class to provide some enhanced functionality, but

that class doesn't support the interface expected by other components within the system that have been running fine all along. So to solve the problem, the idea is that you can put an Adapter class around the new framework class so that the new, improved behavior is supplied via the interface the application expects.

While the discussions in the book center around this type of use for the Adapter pattern, I'd like to suggest that there are still a variety of situations where a *design-time* use of the Adapter pattern can be useful. As most of you probably know from my previous columns, I'm a big proponent of anything that will reduce the future maintenance burden in our applications. By using an Adapter class around different types of components within our applications, the long-term maintenance of those applications can be drastically reduced. In this case, walking hand-in-hand with decreased maintenance cost is increased reusability because some types of components are used throughout a system, and a common interface to those components makes using them in applications more straightforward. That is, each place in the application that uses a "wrapped" component doesn't have to "reinvent the wheel" when it comes to interpreting and/or translating the component's interface.

So, what type of components am I referring to? For starters, just about any ActiveX control is a good candidate for the Adapter pattern. In the VFP world, we've been very lucky in that the product's designers have kept backward compatibility high on their list of priorities. It's rare that a change is made to the interface of one of the native VFP base classes. While we've occasionally seen new methods added to native controls, bugs fixed in the behavior of existing methods, and even new default values for some properties, not much code actually breaks when a new version of VFP is released.

With ActiveX controls, however, the courtesy extended by Microsoft's VFP team isn't quite as common. New controls quite often mean a rewrite of client code before the enhanced features or capabilities can be used. Also, it's not uncommon now for an application to use a number of different ActiveX controls from different vendors. Sometimes, however, it's necessary to switch from using the control provided by Vendor A to a control provided by Vendor B. For example, suppose you're using Vendor A's charting and graphing control to provide a variety of charts in your application. You've got pie charts here, line charts there, and a few hi-lo charts thrown in for good measure. Unfortunately, the first version of your software doesn't support the ability to customize the color for each data series because the control doesn't provide you with that capability. The users aren't happy, and the word from on high is that the next version had better support it. Now what? Well, the obvious first step is to start shopping around for a new charting control. You come across Vendor B's control, and it does everything you need and then some. The end users can go nuts with colors, all of the other features are supplied, and everything's peachy.

Uh-oh. The programmatic interface for Vendor B's control is very different from Vendor A's control. So much for dropping a new control into your application and going home early.

Here's where the Adapter pattern can save your day. If, when initially designing the application, you created a wrapper class around Vendor A's charting control, then your life just got a little easier. All of those places in the application are talking to the charting control via the wrapper class and don't ever deal directly with the underlying control, right? Now all you have to do is create a new wrapper class for Vendor B's charting control. Provided this new wrapper class has the same programmatic interface as the previous one, *none of your client code has to change*. Without that wrapper class, though, it's going to be a long road of changing every place in your application that dealt directly with Vendor A's control, modifying the code to deal with a completely new interface.

Another area where the Adapter pattern can provide similar advantages is with COM (Component Object Model) components. Given all of the emphasis that Redmond is putting on COM objects and using multiple tiers within our applications, you can bet that you're going to continue to see the number of third-party COM and DCOM (Distributed Component Object Model) components increase at a fast pace. Unfortunately, none of those vendors are meeting with their competition to make sure it's easy for us to move from one vendor's component to another vendor's. Given this, it's up to us to keep a layer of abstraction between our client code and those components—an ideal place for the Adapter pattern. In fact, I might even go so far as to say that putting a wrapper class between our client code and *any non-VFP elements* in our projects is a good idea. I've used that approach for a while now with things like Automation servers (such as Excel), third-party function libraries, reporting tools, and even some of my own FLL libraries. It's saved me from big headaches more than once when a change was made in one of those elements and my application needed to adjust accordingly.

An Excel-ent example

To demonstrate the use of an Adapter pattern, I'm going to use a few skeleton classes that might be used to deal with Microsoft Excel as an Automation server. To start with, I need a class that's a wrapper around Excel, but that's not nearly abstract enough for our purposes. Excel is a spreadsheet application, and, contrary to popular belief, there are still a few other spreadsheet applications

out there that some of your clients might actually be using. It makes sense to have an abstract "spreadsheet automation" class that provides the programmatic interface for spreadsheet applications in general. This lets our client code use a subclassed spreadsheet Adapter without regard to which underlying spreadsheet is being manipulated by the Adapter code.

Not only are there common operations to spreadsheet use, but Automation components in general have a few different interaction requirements that are common across all components. For example, anytime you deal with an Automation server, you have to maintain the object reference to the server, close the server when you're finished, and trap errors coming from the server. With all these considerations in mind, the UML (Unified Modeling Language) diagram in Figure 2 shows the type of class hierarchy I have in mind.

In this first class definition, cstAutomation, only the bare minimum interface is provided. This will be the interface that's used across all types of Automation servers (spreadsheets, word processors, and so forth). As you can see, some methods in this abstract class, such as GetApp, SaveOldError, and SetOldError, have been implemented because their implementation will be consistent for all types of Automation servers. The majority of the methods, however, have been defined without implementations. While those methods are required for all subclasses, the implementations must be unique to each application.

```
DEFINE CLASS cstAutomation AS Custom
  *-- Contains the object reference to the
  *-- Automation server application.
  oapp = .NULL.

  *-- Optionally contains the name string of
  *-- the server application to launch
  *-- (i.e., "excel.application").
capplication = ""

*-- Contains the path/filename of the open
*-- document in the parent application.
cfilename = ""

*-- Determines whether or not the application
*-- should be closed when THIS is destroyed.
lcloseondestroy = .T.

*-- Contains the name of the error handler
*-- in place prior to lanching the app.
colderror = ""

Name = "cstAutomation"

*-- Indicates whether the open document in the
*-- application has been saved.
lissaved = .F.

*-- Launches the application, placing a reference
*-- to it in THIS.aApp.
PROCEDURE getapp
  LPARAMETERS tcApplication
  LOCAL llRetVal

  *-- Save the old error handler so we can
  *-- return to it and initialize the
  *-- return value to True.
  THIS.SaveOldError()
  llRetVal       = .T.

  *-- Change the error handler to indicate the
  *-- success of getting an object reference
  *-- to the server application.
  ON ERROR llRetVal = .F.

  *-- Open the server app, placing a reference
  *-- into the oApp property.
  THIS.oApp = CREATEOBJECT(tcApplication)

  *-- Return to the old error handler.
  THIS.SetOldError()

  *-- Send the return value back to the Init()
  *-- so that the calling code knows whether
  *-- or not the object exists.
  RETURN llRetVal
ENDPROC

*-- Save the current error handler into the
*-- .cOldError property so that the calling
*-- method can change it.
PROCEDURE saveolderror
  *-- Save the current error handler into
  *-- the custom .cOldError property.
  THIS.cOldError = ON('Error')
ENDPROC

*-- Sets the error handler back to the original
*-- program (contained in THIS.cOldError).
PROCEDURE setolderror
  LOCAL lcOldError

  *-- Get the old error handler out of the
  *-- custom property.
  lcOldError = ALLTRIM(THIS.cOldError)

  *-- Set the VFP error handler back to
  *-- that program.
  ON ERROR &lcOldError
ENDPROC

PROCEDURE Destroy
  IF THIS.lCloseOnDestroy
    THIS.CloseApp()
  ENDIF
ENDPROC

PROCEDURE Init
  LPARAMETERS tcApplication
  LOCAL lcApplication
```

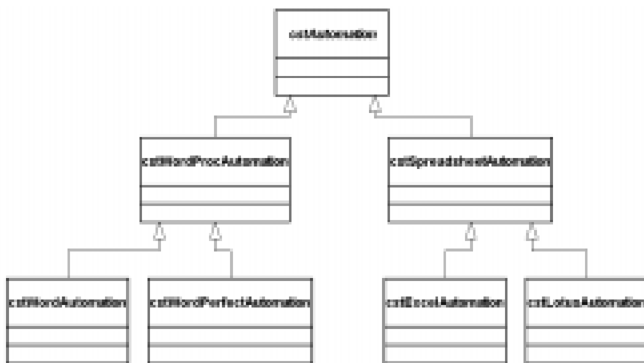


Figure 2. In this class hierarchy, the cstAutomation class provides the interface that's common to all types of Automation servers. Beneath that are abstract class definitions with interfaces for different types of applications, with the implementation-level classes being specific to a certain vendor's applications.

```
    DO CASE
     CASE PCOUNT() = 1 AND ;
      TYPE('tcApplication') == "C"

         lcApplication = ;
           UPPER(ALLTRIM(tcApplication))

     CASE PCOUNT() # 1 AND ;
       NOT EMPTY(THIS.cApplication)

         lcApplication = ;
           ALLTRIM(THIS.cApplication)

     OTHERWISE
        RETURN .F.
    ENDCASE

    RETURN THIS.GetApp(lcApplication)
  ENDPROC

  *--
  *--
  *-- Abstract classes to be implemented
  *-- in subclasses.

*-- This is only a partial class definition!
*-- See this month's Subscriber Downloads
*-- for the complete definition.

  *-- Makes the application
  *-- visible to the user.
  PROCEDURE show
  ENDPROC

  *-- Makes the application non-visible to the user.
  PROCEDURE hide
  ENDPROC

  *-- Closes the OLE Automation client
  *-- application.
  PROCEDURE closeapp
  ENDPROC

ENDDEFINE
```

The next class definition, cstSpreadsheetAutomation, is a subclass of the cstAutomation class that's specific to spreadsheet applications—inserting rows or columns, setting cell values, setting cell formulas, and so on. As with the previous class definition, most of the methods have been left without implementations. Once again, this is because the implementations for performing spreadsheet operations will be specific to each vendor's application (Excel, Lotus, and so forth).

```
DEFINE CLASS cstSpreadsheetAutomation AS ;
               cstAutomation

  *-- Contains an internal object reference to
  *-- the active worksheet.
  oactivesheet = .NULL.
  capplication = ""
  Name = "cstSpreadsheetAutomation"

  *-- Contains the names of all available sheets
  *-- in the open workbook.
  DIMENSION asheets[1]

*-- This is only a partial class definition!
*-- See this month's Subscriber Downloads
*-- for the complete definition.

  *-- Returns a cell's Value contents when passed
  *-- a cell address.
  PROCEDURE getvalue
  ENDPROC

  *-- Sets the passed cell address's Value property
  *-- to the passed value.
  PROCEDURE setvalue
  ENDPROC

  *-- Returns a cell address's Formula property
  *-- when passed a cell address.
  PROCEDURE getformula
  ENDPROC

  *-- Sets the passed cell address's Formula
  *-- property to the passed formula string.
  PROCEDURE setformula
  ENDPROC

  *-- Inserts a row (or rows) above the
  *-- passed row number.
  PROCEDURE insertrow
  ENDPROC

  *-- Inserts a column (or columns) prior to
  *-- the passed column number or range.
  PROCEDURE insertcolumn
  ENDPROC

ENDDEFINE
```

Finally, there's the cstExcelAutomation class. Here's where the implementation for the Adapter is finally provided because the code for performing each operation is unique to Excel's object model. The Adaptee—Microsoft Excel, in this case—can change or be enhanced without all of the client code that uses it being affected. Only this class definition has to change so that the interface that the clients expect can still perform spreadsheet operations, return values to the client, and so on. Furthermore, all the same clients can use a class definition for a different spreadsheet application, such as IBM's Lotus, because the Adapter for Lotus would have the same expected interface.

```
DEFINE CLASS cstExcelAutomation AS ;
               cstSpreadsheetAutomation

  cfilename = ""
  capplication = "excel.application"
  Name = "cstexcelautomation"

  PROCEDURE setvalue
    *: Sets the value for the passed cell address.
    *: The cell address
    *: must come in the R1C1 format.

    *: tuFormula - The value to be placed into the
    *:   specified cell address.
    *: tnCellRow - The row index for the cell
    *:   address whose formula should be returned.
    *: tnCellCol - The column index for the cell
    *:   address whose forumla should be returned.
    LPARAMETERS tuValue, tnCellRow, tnCellCol
    LOCAL llRetVal

    *-- Lots of parameter-checking to do.

    IF TYPE('tuValue') == "L"
      tuValue = IIF(tuValue, "T", "F")
    ENDIF

    *-- Check to make sure that our active
    *-- sheet isn't a Module or Chart sheet.
    IF THIS.IsSheet()
        *-- Set the return value and place
        *-- the formula.
        llRetVal = .T.
```

```
      THIS.oApp.Cells(tnCellRow, ;
         tnCellCol).Value = tuValue
   ELSE
      llRetVal = .F.
   ENDIF

   RETURN llRetVal
   *-- End of method
 ENDPROC

 PROCEDURE insertcolumn
   *-- Inserts one or more columns into the
   *-- active worksheet in the open Excel
   *-- instance. The first parameter is the
   *-- column to be inserted in front of. An
   *-- optional second parameter indicates the
   *-- number of columns to insert, starting
   *-- with the first parameter. As columns are
   *-- inserted, the columns to the right slide
   *-- over toward the right to create the
   *-- new space.

   * tnColumn - The column index number to be
   *  inserted in front of.
   *
   * tnHowMany [optional] - Indicates the number
   * of columns to insert, beginning with tnColumn.
   LPARAMETERS tnColumn, tnHowMany
   LOCAL llRetVal, lnCounter

   *-- Set the return to .T. and
   *-- check to see whether we have a valid
   *-- second parameter.
   llRetVal = .T.
   IF PCOUNT() = 2             AND ;
      TYPE('tnHowMany') == "N" AND ;
      tnHowMany > 1            AND ;
      tnHowMany < XLMAX_AVAILABLECOLS

   ELSE
      tnHowMany = 1
   ENDIF

   *-- Make sure that there's an Excel
   *-- instance, an active workbook and a
   *-- worksheet (rather than a chart or
   *-- module sheet).
   IF llRetVal       AND ;
      THIS.IsSheet()
       *-- Insert the indicated columns (tnHowMany)
       *-- times. Note that the same column is
       *-- inserted in front of each time.
       *-- This is because, as each
       *-- column is inserted, the others will
       *-- slide right to make room for it.
       *-- So, if we have to insert five columns,
       *-- we can just sit on the same
       *-- column and insert at that point
       *-- five times.
     FOR lnCounter = 1 TO tnHowMany
      *-- This line broken for FoxTalk
      *-- formatting.
      WITH THIS.oApp.ActiveWorkbook.ActiveSheet
        .Columns(tnColumn).Insert()
      ENDWITH
     ENDFOR && lnCounter = 1 TO tnHowMany
   ELSE
      llRetVal = .F.
   ENDIF

   THIS.SetSaved(NOT llRetVal)

   RETURN llRetVal
   *-- End of method
 ENDPROC

ENDDEFINE
```

Adapt or perish

I think it's safe to say that H.G. Wells was probably not referring to software when he said, "Adapt or perish, now as ever, is nature's inexorable imperative." It's not likely that your application will suffer nature's wrath and perish just because it doesn't have an Adapter or two. Nonetheless, you might get to a point in the future where your application will have to limp along without updates because new components or objects can't be used in the system without breaking a lot of client code. Look into wrapping an Adapter around those objects, and you'll probably find that making substitutions in your system is much easier. And there's nothing better than easy maintenance, right?

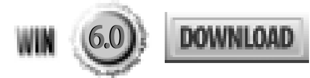
Until next time, don't hesitate to e-mail me at either of the addresses given in my bio if you have comments, suggestions, or questions about the Best Practices column. ▲

07DONNIC.ZIP at www.pinpub.com/foxtalk

Jefferey A. Donnici is the senior Internet developer at Resource Data International, Inc., in Boulder, CO. He was all set to write something about how adaptable he is to changes in his environment, but then his wife couldn't suppress her laughter. Jeff is a Microsoft Certified Professional and a four-time Microsoft Developer Most Valuable Professional. 303-444-7788, fax 303-928-6605, jdonnici@compuserve.com, or jdonnici@resdata.com.

Splitting Up is Hard to Do

Doug Hennig



While they aren't used everywhere, splitter controls can add a professional look to your applications when you have left/right or top/bottom panes in a window. This article implements a splitter using the new OLE drag and drop features added in VFP 6.

SPLITTERS are interesting controls: They don't really have a visual appearance themselves, but they allow you to change the relative size between two or more other controls by adjusting the size of one at the expense of the other. Splitters appear in lots of places in Windows applications; for example, in the Windows Explorer, you can adjust the relative sizes of the left and right panes using a splitter. Splitters can be horizontal (they adjust objects to the left and right) or vertical (they adjust objects above and below), and you could have both types of splitter on the same form. A VFP example of a splitter is in the Class Browser; you can adjust the sizes of the four panes using both horizontal and vertical splitters.

Recently, I needed to add a splitter control to a form I was working on. I looked at the ctSplitter ActiveX control from DBI Technologies (which makes wonderful ActiveX controls, most of which work just splendidly with VFP) but couldn't make it work because it expects to work with windows, and VFP forms aren't true Windows windows (in other words, it isn't DBI's fault). So, I figured that since the Class Browser has a splitter control, and we now have source code for the Class Browser (included in XSOURCE.ZIP in the TOOLS subdirectory of the VFP home directory), I'd just steal, er, borrow the splitter control used in that tool. However, after looking at the code, I ran into a couple of issues: I don't like the visual behavior of the control (the other objects on the form become invisible while you drag the splitter), and it's so specific to the Class Browser that making it work with my form (or even better, creating a reusable tool out of it) would be more work than it's worth.

So, I was stuck with building my own. After going down the same path as the Class Browser splitter for a while (trapping the MouseDown event of a shape object and then using DOEVENTS to allow the user to drag the shape around) and not enjoying where I was going, I decided to step back and think about things a bit. It occurred to me that the behavior I wanted was to drag a shape on a form and have it adjust the size and position of other controls as I did so. My first thought was to use VFP drag and drop, but I quickly discarded that idea when I remembered that the source object (the thing being dragged) doesn't get any events during the drag operation; instead, the target objects (the things being dragged over) get a DragOver event. I didn't want to put code into the DragOver event of every object on a form (which would be required so the shape could be dragged anywhere), so I discarded that idea. Then I remembered that with OLE drag and drop, new to VFP 6, the source object *does* receive events when it's dragged over a target object. Looks like we're on the right track so far.

Not having used OLE drag and drop before, I turned to my main resource for information on VFP—a book whose name I won't mention because I was the technical editor, *FoxTalk* editor Whil Hentzen is the publisher, and the authors are friends of mine (okay, it's the *Hacker's Guide to VFP 6* <g>). The event that seemed most promising was OLEGiveFeedback, which is fired in the source object after the OLEDragOver event of the target object is fired. This seemed like the perfect place to adjust the sizes of the objects being managed by the splitter. That led to another complication: Only objects that have the OLEDropMode property set to 1 (Enabled) would cause OLEGiveFeedback to fire. So, I was back to setting this property to .T. for each object on the form, right?

Well, a thought occurred to me. What if I were to place an invisible object on the form that covers everything else and have it be the target for the OLE drag and drop by setting its OLEDropMode to 1? That proved to be the winning strategy.

Okay, enough of following my twisted thought processes. Let's get into some code!

SFSplitter

The first class I created is SFSplitter, contained in SFSPLITTER.VCX. This class, which is based on SFShape (our shape base class in SFCTRLS.VCX), won't be used directly but will serve as the parent class for horizontal and vertical splitter classes. Although the shape will be invisible at runtime, it's easier to work with if it has a visual appearance at design time, so I left the BorderStyle property at the default and made the Init method set it to 0 (none) at runtime. If it's invisible, how does the user know a splitter is available? First, we'll drop it between two controls (such as list boxes, TreeViews, ListViews, and so on) that might logically be adjusted with a splitter, and second, we'll set the MousePointer property so when the mouse is over the control, it looks like a splitter. We won't

set MousePointer in this class, because which pointer to use depends on whether we have a horizontal or vertical splitter, so we'll do that in a subclass.

As I mentioned, the Init method sets BorderStyle to 0 then it adds an SFSplitterCover object (which I'll discuss later) to the form. Notice that it uses the new NewObject method (added in VFP 6) of the form and specifies its own ClassLibrary property as the location of the SFSplitterCover class. It also assigns a unique name to the object if one wasn't specified in the cSplitterCoverName property and stores the assigned name in this same property (we need to know the name because we'll be talking to it later). Because SFSplitter adds this object (which collaborates with it during the drag operation) to the form, this class doesn't require that you do anything to the form or other objects to use it. That's my idea of a reusable tool! Here's the code for Init:

```
local lcName
with This

* Make the border invisible.

  .BorderStyle = 0

* Add an SFSplitterCover object to the form.

  lcName = iif(empty(.cSplitterCoverName), sys(2015), ;
    .cSplitterCoverName)
  .cSplitterCoverName = lcName
  Thisform.NewObject(lcName, 'SFSplitterCover', ;
    .ClassLibrary)
endwith
```

To enable OLE drag and drop, I set the OLEDragMode property to 1 (Automatic) so OLE drag and drop starts as soon as the user starts dragging the shape. The three OLE drag and drop events we're interested in are OLEStartDrag, which fires when the drag starts, OLEGiveFeedback, which fires when the object is dragged over a target, and OLECompleteDrag, which fires when the drag operation is done. Here's the code for OLEStartDrag:

```
lparameters toDataObject, ;
  tnEffect
This.StartMoving()
```

Here's the code for OLEGiveFeedback:

```
lparameters tnEffect, ;
  tuMouseCursor
tuMouseCursor = This.MousePointer
This.MoveSplitter()
```

Here's the code for OLECompleteDrag (unlike its name suggests, OLE drag and drop is actually kind of fun <g>):

```
lparameters tnEffect
This.DoneMoving()
```

With the slight exception of OLEGiveFeedback, each of these events simply calls a custom method of the form (I'm taking to heart Steven Black's advice that "events call methods"). OLEGiveFeedback also sets the mouse pointer for the drag operation (passed by reference to this event in the tuMouseCursor parameter) to the same value the class itself uses, so we get a consistent look as the splitter is dragged.

StartMoving, called from OLEStartDrag, makes the SFSplitterCover object it added to the form visible, the same size as the form, and "above" everything else on the form so it will be the sole target object for the drag operation. StartMoving also calls the SetStartingPositions method, which isn't implemented in this class because it's specific to the type (horizontal or vertical) of splitter we'll use.

```
local loCover

* Size the SFSplitterCover object to cover the entire
* form, make it visible, and move it to the top of the
* Z order.

loCover = evaluate('Thisform.' + This.cSplitterCoverName)
with loCover
  .Width   = Thisform.Width
  .Height  = Thisform.Height
  .Visible = .T.
  .ZOrder(0)
endwith

* Call a method to set the starting positions.

This.SetStartingPositions()
```

MoveSplitter, called from OLEGiveFeedback, is an abstract method because its behavior depends on whether the splitter is horizontal or vertical. I also added an abstract AfterMoveSplitter method that can be used in instances or subclasses of a splitter to perform additional operations.

DoneMoving, called from OLECompleteDrag, simply hides the SFSplitterCover object when the drag operation is done:

```
local loCover
loCover = evaluate('Thisform.' + This.cSplitterCoverName)
loCover.Visible = .F.
loCover.ZOrder(1)
```

Two other methods, GetFirstObject and GetObjectNames, aren't called by anything in SFSplitter but will be called from subclasses. These methods accept a comma-delimited list of names and return the first name in the list and populate an array with the names, respectively. We'll see how they're used later.

SFSplitterCover

Like SFSplitter, SFSplitterCover is based on SFShape. Its purpose is very simple: Act as a target for an OLE drag and drop operation of an SFSplitter object. Its OLEDropMode property is set to 1 (Enabled) so it can be a target, and its BorderStyle is set to 0 so it doesn't have any

visual representation. Its OLEDragOver event, fired when the SFSplitter object is dragged over it, simply stores the X and Y coordinates passed to it to custom nXCoord and nYCoord properties:

```
lparameters toDataObject, ;
    tnEffect, ;
    tnButton, ;
    tnShift, ;
    tnXCoord, ;
    tnYCoord, ;
    tnState
This.nXCoord = tnXCoord
This.nYCoord = tnYCoord
```

SFSplitterH and SFSplitterV

SFSplitterH is a horizontal splitter subclass of SFSplitter. The way it works is that one or more objects to the left of the splitter are considered to be anchored at their left edges, so only their widths will be adjusted as the splitter is moved. One or more objects to the right of the splitter are anchored at their right edges, so both their Left and Width are adjusted accordingly. The net effect is that as the splitter is moved to the right, the left objects get wider at the expense of the right objects, and vice versa as the splitter is moved to the left. The relative distances of the left objects from the left edge of the form, the right objects from the right edge of the form, and the left and right objects from the splitter and each other stay fixed.

I set the Height property of SFSplitterH to 200, Width to 10 (you'll likely change the Height of an instance of this class to match the objects you're splitting but probably won't need to change the Width), and MousePointer to 9 (Size W E). Only two methods are overridden in this class. The first is SetStartingPositions:

```
local lcObject, ;
  loObject
with This

* Save the distance between the left object
* and the splitter.

  lcObject = .GetFirstObject(.cLeftObjectName)
  loObject = evaluate('.Parent.' + lcObject)
  .nLeftObjectSpace = .Left - loObject.Width

* Save the distance between the right object and the
* splitter and the object's width.

  lcObject = .GetFirstObject(.cRightObjectName)
  loObject = evaluate('.Parent.' + lcObject)
  .nRightObjectLeft  = loObject.Left - .Left
  .nRightObjectWidth = loObject.Left + loObject.Width
endwith
```

This code calls the GetFirstObject method, passing it the value of the custom cLeftObjectName property. The idea of cLeftObjectName is that you'll enter a comma-delimited list of the names of the objects to the left of the splitter control. SFSplitterH assumes that while there might be more than one "left" object being managed by the control, they all have the same Width, so only the first one is used. nLeftObjectSpace contains the distance between the right edge of the left controls and the left edge of the splitter (not quite true because we aren't taking the Left property of the left objects into consideration, but since we won't be changing that property of these objects, we can factor that out of the equation).

A similar calculation is done for the first right object listed in cRightObjectName, with the distances between it and the splitter and it and the right edge of the form stored in nRightObjectLeft and nRightObjectWidth, respectively. Why are these values calculated when a drag is started rather than in the Init of the class? You might programmatically assign the names in cLeftObjectName and cRightObjectName rather than doing it in the Property Sheet. Why are names listed in the properties rather than storing object references to them? It's easier to specify the objects by listing them in the Property Sheet rather than having to write code that places object references somewhere (likely an array)—plus it avoids the issue of dangling object references caused by not destroying extra references to objects.

The other method with code in SFSplitterH is MoveSplitter, which is called from OLEGiveFeedback when the splitter is dragged over the target.

```
local loCover, ;
  lnXMovement, ;
  lcObject, ;
  loObject1, ;
  loObject2, ;
  lnWidth1, ;
  lnLeft, ;
  lnWidth2, ;
  laObjects[1], ;
  lnObjects, ;
  lnI, ;
  loObject
with This

* Get the current X coordinate for the drag from the
* SFSplitterCover object.

  loCover = evaluate('Thisform.' + .cSplitterCoverName)
  lnXMovement = loCover.nXCoord

* Get object references to the left and right objects
* and adjust their width and left/width properties,
* respectively, as long as we don't make one of them
* smaller than the minimum width. Then move ourselves
* to the X position.

  lcObject  = .GetFirstObject(.cLeftObjectName)
  loObject1 = evaluate('.Parent.' + lcObject)
  lcObject  = .GetFirstObject(.cRightObjectName)
  loObject2 = evaluate('.Parent.' + lcObject)
  lnWidth1  = lnXMovement - .nLeftObjectSpace
  lnLeft    = lnXMovement + .nRightObjectLeft
  lnWidth2  = .nRightObjectWidth - lnLeft
  if lnWidth1 >= .nLeftMinWidth and ;
    lnWidth2 >= .nRightMinWidth
    lnObjects = .GetObjectNames(.cLeftObjectName, ;
      @laObjects)
    for lnI = 1 to lnObjects
      loObject = evaluate('.Parent.' + laObjects[lnI])
      loObject.Width = lnWidth1
    next lnI
    lnObjects = .GetObjectNames(.cRightObjectName, ;
      @laObjects)
    for lnI = 1 to lnObjects
      loObject = evaluate('.Parent.' + laObjects[lnI])
      loObject.Width = lnWidth2
      loObject.Left  = lnLeft
    next lnI
    .Left = lnXMovement
```

```
* Call a hook method so a subclass could do something
* else (like moving other objects).

    .AfterMoveSplitter(lnXMovement)
  endif lnWidth1 >= .nLeftMinWidth ...
endwith
```

This code gets the value of the nXCoord property of the SFSplitterCover object; this property was set to the current X coordinate of the drag operation by the OLEDragOver method of that object. It then determines how much to adjust the width of the left objects and the left positions and widths of the right objects. If the width of either the left or right objects would be made too small (enter the minimum values into the nLeftMinWidth and nRightMinWidth properties), nothing is moved or adjusted. Otherwise, each of the left and right objects is adjusted, and the Left property of the splitter itself is changed to match the mouse position. Finally, the abstract AfterMoveSplitter is called with the mouse position as a parameter so an instance or subclass of the splitter could do some additional tasks.

SFSplitterV is a vertical splitter subclass of SFSplitter. I won't show its code here because it's almost identical to SFSplitterH, except instead of managing the Left and Width properties of left and right objects, it manages the Top and Height properties of objects above and below the splitter. Enter the names of the objects to manage in the cTopObjectName and cBottomObjectName properties and their minimum heights in the nTopMinHeight and nBottomMinHeight properties. The Height and Width of this class are reversed from SFSplitterH so it appears as a horizontal shape, and its MousePointer is 7 (Size N S).

As an extra goody, I've created BuilderD builders (see my March 1999 column, "Building Builders with BuilderD," for information on BuilderD) for both SFSplitterH and SFSplitterV. Simply open the BUILDERD table in the WIZARDS subdirectory of your VFP home directory and append from the BUILDERS table included with this month's Subscriber Downloads at www.pinpub.com/foxtalk.

Sequence of events

Just so it's clear what happens when, here's the sequence of events that fire. When the SFSplitter subclass is instantiated, it adds an SFSplitterCover object to the form, but this object is invisible for now. When the user starts dragging the splitter object, its OLEStartDrag event fires, which makes the SFSplitterCover object visible (although it's transparent and has no border, so it doesn't really appear to be visible) and the topmost object. As the splitter is dragged, the OLEDragOver event of the SFSplitterCover object fires, which stores the current mouse location. Then the OLEGiveFeedback event of the SFSplitter object fires, which moves all of the managed objects to their new locations. When the user lets go of the mouse, the OLECompleteDrag event of the SFSplitter object fires, which makes the SFSplitterCover object invisible again and moves it to the back of the Z order.

Sample form

Let's check out these splitter controls. To make it interesting, let's put both vertical and horizontal splitters on a form *and* let's throw an SFResizable object (adjusts the sizes of controls as a form is resized, as discussed in my December 1998 column, "Mining for Gold in the FFC") on the form just for grins.

TEST.SCX contains four ListBox objects, named List1 to List4 (see **Figure 1**). The Resize method of the form calls the AdjustControls method of an SFResizable object named oResizer; this object has its properties set so List4 is resized in both directions, List3 is only resized vertically, List2 is only resized horizontally, and List1 isn't resized at all. An SFSplitterH object sits between the left and right list pairs and has cLeftObjectName set to "List1, List3," cRightObjectName set to "List2, List4," and both nLeftMinWidth and nRightMinWidth set to 100. An SFSplitterV object sits between the top and bottom list pairs and has cTopObjectName set to "List1, List2," cBottomObjectName set to "List3, List4," and both nTopMinHeight and nBottomMinHeight set to 40. The DoneMoving method of both splitters calls the base behavior with DODEFAULT(), then calls the Reset method of the SFResizable object so it can adjust itself to new object sizes.

Figure 2 shows the results of running this form and using both the horizontal and vertical splitters to resize the list boxes. Try resizing the form and see how that affects the appearance of the objects.

Conclusion

Splitter controls add a professional polish to an

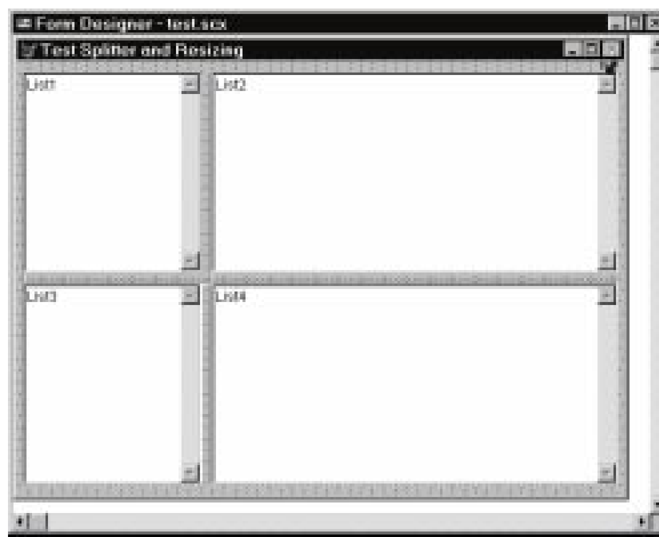


Figure 1. TEST.SCX.

application because users are starting to expect that they can adjust the sizes of left/right or top/bottom panes in a window. The reusable classes presented in this article are self-contained: Just drop them on a form, set some properties, and your users suddenly have new freedom to rearrange your forms as they see fit. ▲

DOWNLOAD 07DHENSC.ZIP at www.pinpub.com/foxtalk

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. 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 the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). dhennig@stonefield.com.

Figure 2. TEST.SCX after using both splitters.

Now You See It . . .

Continued from page 7

Paul: I see you're using the Zorder() to force the grid to cover up anything on the form that lies in the area below the closed control.

Andy: Yes, you get some weird effects without it, too. Notice that I did put in a hook so that if I ever get around to using a resizable container, it will get called here and in the DoPop(), which closes the grid up!

```
*** DOPOP METHOD*** Close the grid and hide it
WITH This
    *** If editing, call the Update method first
    IF .lEditing
        .DoUpdate()
    ENDIF
   *** Set Flags
   .lIsDropped = .F.
   *** Hide Grid
   .lblPrompt.Visible = .F.
   .grdItms.Visible = .F.
   IF ! .lSelected
      .txtSch.Value = .cSchStr
      KEYBOARD '{END}'
   ENDIF
   .txtSch.SetFocus()
    *** If in a resizable container,
    *** resize the Container first
    IF LOWER( .Parent.ParentClass ) = "xcntstdautosize"
        .Parent.DoShrink()
        *** Make background transparent
      .BackStyle = 0
    ENDIF
   *** Send to back and reset button caption
   .Zorder(1)
   .cmdDrop.Caption = CHR(234)
ENDWITH
```

Paul: You know, this code looks sort of familiar to me somehow.

Andy: And so it should; you wrote most of it in your cntDroppable class—all I did was modify it to suit my needs in this specific instance.

Paul: I didn't realize that the title referred to the code as well as to the principles <g>. I presume the rest of the code (which, by the way, is included—together with a demonstration form—in the Subscriber Downloads at www.pinpub.com/foxtalk) is pretty standard.

Andy: Yes indeed. There's some nice (well, I think so) handling of the incremental search and some pretty neat stuff for making the grid editable one row at a time, but none of that is my code, and it's all very well-documented in the example.

Paul: Oh, a word of warning. There seems to be a bug in the handling of the keyboard shortcut *{CTRL+DnArrow}* for opening the grid. Unless you actually recompile the class library specifically on your own machine, it doesn't seem to work when you just run the demo. To date, we have no idea why this is so, it just is.

*[**Andy:** A couple of acknowledgements are due here. First, to Tamar Granor for publishing the QuickFill methodology used in the Search text box, and second to Marcia Akins for all of her assistance with the Data Entry and Ordering sections for the grid. Thank you so much, ladies!]* ▲

DOWNLOAD CBOGRID.ZIP at www.pinpub.com/foxtalk

Andy Kramek is a long-time FoxPro developer, FoxPro MVP, independent contractor, and occasional author, currently of no fixed abode, who works in Buffalo, NY. andykr@compuserve.com.

Paul Maskens is a VFP specialist and FoxPro MVP who works as a programming manager for Euphony Communications Ltd. He's based in Oxford, England. pmaskens@compuserve.com.

I Was Framed!

Paul Maskens and Andy Kramek

When starting out to build an application in Visual FoxPro, you have three choices: Build everything you need yourself, use FoxPro's supplied classes as the basis, or use a commercial framework. Paul and Andy have some thoughts on the merits of each approach …

Andy: This month, I'd like to get your views on something really basic to working with VFP. Should you use a framework for your applications, and if so, how should you decide what to use?

Paul: Wow! That's a big question. I think we'd better begin by ensuring that we understand what we're talking about here. You can't build *anything* in an object-oriented language without some sort of framework (even if the framework is merely your class library), so perhaps your question should be broken down into two parts: "What constitutes a framework in VFP?" and "Should you try to build your own or use one of the commercially available frameworks?"

Andy: Okay, I can live with that—so what actually *is* a framework?

Paul: I suppose the most basic definition is that a framework is a set of tools that allows you to build applications without having to worry about the basic functionality required to actually manage the tables, forms, and toolbars. It's also important to differentiate between a framework and an application generator. The former provides you, the developer, with a set of tools upon which you can build, while the latter actually creates your application for you.

Andy: Let's leave application generators out of it, shall we? That's still quite a wide ranging definition for a framework that you've given there. Can we try to crystallize things a little? It seems to me that there are three fundamental things that one needs to consider when designing any application. First, there are the "system-level" services that you need (paths, environment settings, and so on); second, there are the "data handling" considerations (DBC, tables, connections); and finally, there's the user interface (the actual forms and all of the things that make the application work).

Paul: The UI is definitely application-specific, and while you might well include UI components in the framework, the creation of UI tools is, I think, the key difference between a framework and an application generator. After all, you can't actually build a "generic" application, even for something as simple as an address book! Each application will have its own requirements—or business rules, if you prefer—and they can't be generalized. The other stuff should definitely be handled in the framework.

Andy: That seems reasonable to me. But from what you say, it *is* reasonable that the framework includes a set of UI components. Presumably, these would only be the classes that you'd use when building an application—the forms, text boxes, and so on.

Paul: Yes, that would, I feel, be an absolute requirement because in constructing the other components of the framework, you're going to have to make *some* assumptions about the way the UI is going to interact with the framework.

Andy: A-ha! Now we're getting to it! This is the root of my initial question. How can we identify those assumptions that must be made and those that should be left to the application developer?

Paul: The rule is, I think, that the framework should define abstract behaviors, not concrete implementations.

Andy: Sounds impressive! But what on earth do you mean?

Paul: To take an example, consider the combo grid that we discussed in our article, "Now You See It, Now You Don't," elsewhere in this issue. That's a class that encapsulates a set of behaviors and that has some clearly defined rules, but it doesn't actually do anything by itself. To make it work, you have to supply the additional things it needs (the table, the number and ControlSources for the columns, and so on). In this sense, it is, itself, merely a behavior and not an implementation.

Andy: Okay, I follow you. So if we'd designed the combo grid to work only with, say, a three-column look-up table, it would have been too specific to really qualify as a framework tool, then? Or would that merely be another rule for the class?

Paul: Hmmm! You could argue that one either way, I suppose. On balance, I'd say that a component that defines a rule like that isn't really generic enough to be part of a framework. In fact, that should be one of the fundamental rules of any framework—that it shouldn't assume how any of its components are going to be implemented.

On the other hand, it could be a control that's part of my toolbox—one that I found useful for one application and have kept because I know that, with work, it *could* be generic. Sometimes (do as I say, not as I do) there are occasions where I build a class that's used many times in one application, and I feel it *ought* to be reusable. If only I'd thought about the generic case when building it and taken the extra time to make a generic version.

Andy: Haven't you just contradicted yourself there? A moment ago you said that you had to make *some* assumptions about how the framework would interact with the UI; now you're saying that you should *not* make assumptions about how the components should be implemented.

Paul: There's a subtle difference there, Andy. You have to make assumptions about how the framework will *interact* with the UI—you couldn't program anything otherwise! What you have to do is ensure that you don't make assumptions about *how* the UI is going to be used. One of the components that you might want in your framework is some sort of form manager. Now, you could create a form manager that will work perfectly well with any form once it's up and running but that can only create forms that are defined as either classes or SCX files, but not both. That would be a bad framework because you're defining how the application is to be constructed in addition to defining how it will interact with the framework once it's been built.

Andy: Got it! So I guess that the real question now is: How *should* the framework be constructed? Most of the frameworks I've ever looked at use some sort of "master" class—usually referred to as the "Application Object" (and typically instantiated as "goApp"), which seems to handle both the initialization of the application and provide a link to a whole bunch of standard functions and services. Is this a good starting point?

Paul: Well, it's pretty standard, but I'm not totally convinced that it's essential or even desirable. As far as I can see, running everything through an Application Object imposes an additional layer of referencing. I'm not quite sure what the benefits are supposed to be, either—especially if the additional functionality is already defined in classes (for instance, Environment Manager, Form Manager, Data Manager, Security Manager). You end up having to either add objects based on these classes directly to the Application Object or holding references to them in the Application Object.

Andy: Interesting! I've never used an Application Object either, but I see what you mean. If I were to use my Data Manager with an Application Object instead of instantiating it directly, I'd have to do this:

```
goApp.AddObject( 'oDatMgr', 'xDatMgr' )
```

and then reference it through goApp every time I wanted to use it:

```
goApp.oDatMgr.Do( 'Query', 'curJunk' )
```

instead of what I do now, which is to instantiate the manager directly and hold its reference as a "system-level" variable:

```
goDatMgr = CreateObject( 'xDatMgr' )
goDatMgr.Do( 'Query', 'curJunk' )
```

Paul: Precisely. Of course, you could always grab a reference locally anyway:

```
loDatMgr = goApp.DatMgr
```

But another drawback is that you can't easily test for the existence of your object when it's contained—using VARTYPE() won't give you a reliable response when the object is referenced indirectly.

Andy: Good point! I suppose you'd then have to add an Access Method to the Application Object for each property that references one of its contained objects? Then you could test for the existence of the object, and create it if it doesn't already exist—like this:

```
IF VARTYPE( This.oDatMgr ) # 'O'
   This.AddObject( 'oDatMgr', 'xDatMgr' )
ENDIF
RETURN .T.
```

Paul: Aaaargh! No! I wouldn't do that in an Access method. Remember, an Access method fires *every* time you try to address the object. This would fire every time you tried to set a property or call a method of your Data Manager. While this is, in many ways, "proper" OOP (it hides the management of the object and the access to it), in the real world, this might *not* be a sensible implementation given the number of times you're likely to call things from it, even allowing for the fact that VARTYPE() is quick—you pay a speed penalty.

To continue on the POOP track for a moment longer: One requirement that I hope will encounter no argument is that whatever you use as your framework has to be

efficient. Also, it has to be flexible enough to build the applications you need. There will be a trade-off between flexibility and speed.

Andy: You're absolutely right! In my opinion, one of the hardest things to define is when it's better to give up flexibility in the interest of performance, and when it's better to take the hit! So in this case, we should really give up the flexibility and rely on the error handler?

Paul: Well, maybe there are other ways of handling this particular issue. After all, once created, your objects shouldn't get released "accidentally," so a single test in the Startup() procedure ought to do the trick. But we're getting off the point a bit here (again <g>). The issue is still whether an Application Object can deliver any real benefits for the overhead that it imposes.

Andy: About the only other thing I can think of is that it does ensure that all of your system objects are maintained at the same scope—which must ease the garbage collection issues. When you release the Application Object, it will release all of its contained objects automatically. It gives you an "Auto-Release" facility.

Paul: Is this good? What happens if the Application Object *does* get trashed or released inappropriately? You could be in serious trouble right there! Especially if other objects are relying on its presence.

If all of the other service objects (is that a good term?) were contained in or referenced by the Application Object, releasing or destroying the Application Object would stop everything.

Although, of course, you should never program things so that an object relies on the internal functioning of another object anyway. But calls to the Application Object really should be wrapped in a test to ensure that, at the very least, it's still available—its own error handling must be sufficiently sophisticated to deal with errors arising from its own internal complexity. That's another overhead right there.

Andy: We seem to be in agreement here—the decision to use an Application Object as the basis of your framework isn't quite as simple as it appears at first glance.

Paul: Definitely not! Even if you use global variables, for example, would you still want to check that goDatMgr is an object? If so, the overhead of using an Application Object vs. public variables to hold the reference to the service objects is an issue of the amount you have to type, isn't it?

Andy: To be honest, I don't agree. As you said a moment ago, I tend to rely on a single test at startup and then "assume" that the object is still there. Of course, I'm pretty strict about my use of Public variables and their naming, so it's never (yet <g>) been an issue for me in an application.

Paul: Thinking back to the early days of FoxPro 3, when we were all learning, I saw a proposed application object that had a form manager, toolbar manager, and menu manager built in—not using separate objects but using methods of the Application Object. An application was built by setting properties of an Application Object subclass for caption, menu program name, toolbar associations with forms, and so on.

Although not an application generator, that does seem to me to be more application-specific rather than generic. Would you agree?

Andy: Totally! I think it was probably the inflexibility of that approach that turned me away from the concept of an Application Object in the beginning. It struck me at the time that by keeping these objects as separate classes, I could radically modify behavior by simply swapping the classes on which the various objects were based.

Paul: Agreed. I see it as a constraint on the kind of application that you're going to build. For example, it's presupposed that you wanted modeless forms, toolbars, and a menu. If you didn't want that (for whatever reason), then you either had to remove functionality from the Application Object or carry around a lot of redundant code. I'd far rather work with a collaboration of objects, which are assembled using composition, than with one object that tries to do everything with goApp methods and properties.

Andy: That takes us back to the requirement that our framework has to be generic—it's far easier to be generic with a collection of collaborating objects, providing that the public interfaces of the different classes are consistent with each other, than with a single monolithic object. The one thing we can be absolutely certain of in these days is that our requirements *will* change!

Paul: Moreover, I don't want the framework to constrain the way I develop. It has to provide a set of services and interface tools that I can use as a foundation in building an application, but I don't want it to only have one way of building an application. I wouldn't want it to only work with views, or with data classes but not with views at all, or only with business objects.

Andy: I think you've hit on a key point there, my friend. The role of a framework is to provide *services* to the

application developer. The way in which those services are used is strictly the province of the developer. Oh, and *services*, by definition, have nothing to do with the business problems that the application is trying to address! In other words, the framework is there to remove the necessity to program standard functionality over and over again.

Paul: A-ha! That's a very important point. As soon as you start addressing business problems in a framework, you begin limiting its reusability and flexibility. Maybe that's why we have such a choice of frameworks, from the VFP Foundation classes (with VFP 6 only), the various commercial frameworks, all the way through to application generators. Of course, we can build our own, too.

I also think there's something implied in the two words we're using: framework and foundation.

Andy: Would you care to elaborate?

Paul: Well, a foundation is something you build on, right? It doesn't define *what* you're going to build, or how you do it, does it? So contrast that with a framework.

Andy: Okay, I think I see what you're getting at—the foundation is merely the platform on which you build, and in software that would translate to your database schema and your development environment. But the framework has to provide the support on which you hang the application—or, if you prefer a more biological metaphor, the skeleton around which the flesh is added. In that sense, the framework must define the size and shape of the finished article, if not the actual appearance.

Paul: Yes, this is why I think of a framework (fairly or not) as more restrictive. Think about building a skyscraper and a timber-frame house. They use different frameworks, different architectures, and different building methods with different size teams.

Andy: Yes, I see what you mean. You need different frameworks for different types of buildings. But can you really carry that analogy right over into the software world?

Paul: I know it's not a true analogy, if only because a building's framework is actually part of its design. But I'm wondering if that perception of a framework (more like a "cage") might explain the reluctance I've seen among colleagues to use formal frameworks. I've heard many objections raised: learning time, support, documentation quality, limitations, feelings of constraint, unwillingness to rely on other people's code, availability of fixes and upgrades, availability of new versions as new VFP versions are released.

Andy: All of those objections will apply to any framework—whether you build it yourself or buy a third-party one. It has to be well-documented, easily maintainable, and extensible, or it's no use at all! As for learning time, I suppose you have to determine whether it's going to be more cost-effective to write your own or to learn how to use someone else's. Whatever you use, of course, it must be both robust and bug-free!

Paul: Certainly, the goal of reuse requires that the classes that you use be robust, and in order to use them at all, they have to be well-documented. Talking to several contractors, I get the feeling that in being willing to use other people's code, I'm the odd one out. Yet when working on a team on a project, you're always relying on other people's code.

Andy: Agreed, but when the code is being written by the team in which you're participating, you can at least have some input into *how* that code is written, as well as knowing something about *why* it's written in a particular way, of course. That does make a difference, I think.

Paul: Okay, then, a loaded question: Would you use a third-party framework that didn't provide source code?

Andy: Easy question. No, I wouldn't.

Paul: Would you use VFP components that didn't provide source code?

Andy: Generally speaking, from choice, I'd have to say that I wouldn't—if only because I usually want a particular "look and feel" to my applications.

Paul: So what about using ActiveX controls, or COM objects? They don't have source code.

Andy: Well . . . that's a bit different <g>. These things, by definition, aren't going to be an integral part of my framework. They exist as standalone objects that provide specific functionality. You either use them, or you don't.

Paul: But once you use products, they become a part of your development environment. You become committed to them if you have to maintain and support your programs. You have to have confidence in the tools and components you use in your development. I think you're using the ability to look at and even fix the code of your third-party tools as a kind of insurance.

Andy: No, they become part of the application, but not part of either my framework or even my "development environment." I can (and do) make use of external controls in my application—an ODBC driver, for example. It's merely something that I need in order to make my application work with an external data source. I'd like to think that if the driver weren't available, my application could still work, just not with an external data source. My framework has to accommodate their use, but it doesn't have to include them!

Paul: Well, part of my development environment is CEE. Admittedly, some of its functions were added to VFP 5, but I still find it very useful for adding variable definitions (Alt+6 to create a local variable declaration for the variable highlighted in the editor). I was considering using XiLights in VFP 3 to get color syntax coding—but MS added the feature to VFP 5 before I got around to buying XiLights. I'd still want to use XiLights with FPW2.6, though—I like the syntax highlighting.

Oh, was I supposed to be making a point? Okay, development environments change. Adopting a framework that's inflexible might inhibit that change.

Andy: I'm not sure that this is relevant, Paul <s>. Your development environment has little to do with your framework as we've discussed it. I think that what we've arrived at the position that the role of the framework should be to provide a set of tools that deliver standard functionality (or services) without attempting to address specific business problems. There are many ways of doing this, and you need to find the one that's most comfortable for your own development style and needs.

Paul: Okay, I can live with that—but next month, we'll look in more detail at what should be in this sort of framework. ▲

Andy Kramek is a long-time FoxPro developer, FoxPro MVP, independent contractor, and occasional author, currently of no fixed abode, who works in Buffalo, NY. andykr@compuserve.com.

Paul Maskens is a VFP specialist and FoxPro MVP who works as a programming manager for Euphony Communications Ltd. He's based in Oxford, England. pmaskens@compuserve.com.

Downloads

July Subscriber Downloads

- CBOGRID.ZIP—Source code described in Paul Maskens and Andy Kramek's article, "Now You See It, Now You Don't."
- 07DONNIC.ZIP—Source code described in Jefferey Donnici's article, "Best Practices: Seeing Patterns: The Adapter."
- 07DHENSC.ZIP—Source code described in Doug Hennig's article, "Reusable Tools: Splitting Up is Hard to Do."

Extended Articles

- 07FALINO.HTM—Jim Falino's article, "Hacking the SQL Server Upsizing Wizard."
- 07FALINO.ZIP—Source code described in Jim Falino's article, "Hacking the SQL Server Upsizing Wizard."
- 07HENTZ.HTM—Whil Hentzen's article, "VB for Dataheads: Real-World Usage of ADO in VB."

FoxTalk Subscription Information:
1-800-788-1900 or http://www.pinpub.com

Subscription rates:

United States: One year (12 issues): $179; two years (24 issues): $259
Canada:* One year: $194; two years: $289
Other:* One year: $199; two years: $299

Single issue rate: $17.50 ($20 in Canada; $22.50 outside North America)*

European newsletter orders:
Tomalin Associates, Unit 22, The Bardfield Centre,
Braintree Road, Great Bardfield,
Essex CM7 4SL, United Kingdom.
Phone: +44 1371 811299. Fax: +44 1371 811283.
E-mail: 100126.1003@compuserve.com.

Australian newsletter orders:
Ashpoint Pty., Ltd., 9 Arthur Street,
Dover Heights, N.S.W. 2030, Australia.
Phone: +61 2-9371-7399. Fax: +61 2-9371-0180.
E-mail: sales@ashpoint.com.au
Internet: http://www.ashpoint.com.au

* Funds must be in U.S. currency.

Editor Whil Hentzen; **Publisher** Robert Williford;
Vice President/General Manager Connie Austin;
Managing Editor Heidi Frost; **Copy Editor** Farion Grove

Direct all editorial, advertising, or subscription-related questions to Pinnacle Publishing, Inc.:
1-800-788-1900 or 770-565-1763
Fax: 770-565-8232

Pinnacle Publishing, Inc.
PO Box 72255
Marietta, GA 30007-2255

E-mail: **foxtalk@pinpub.com**

Pinnacle Web Site: **http://www.pinpub.com**

FoxPro technical support:
Call Microsoft at 425-635-7191 (Windows)
or 425-635-7192 (Macintosh)

FoxTalk (ISSN 1042-6302) is published monthly (12 times per year) by Pinnacle Publishing, Inc., 1503 Johnson Ferry Road, Suite 100, Marietta, GA 30062. The subscription price of domestic subscriptions is: 12 issues, $179; 24 issues, $259. **POSTMASTER:** Send address changes to *FoxTalk*, PO Box 72255, Marietta, GA 30007-2255.

Copyright © 1999 by Pinnacle Publishing, Inc. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever (except in the case of brief quotations embodied in critical articles and reviews) without the prior written consent of Pinnacle Publishing, Inc. Printed in the United States of America.

Brand and product names are trademarks or registered trademarks of their respective holders. Microsoft is a registered trademark of Microsoft Corporation. The Fox Head logo, FoxBASE+, FoxPro, and Visual FoxPro are registered trademarks of Microsoft Corporation. *FoxTalk* is an independent publication not affiliated with Microsoft Corporation. Microsoft Corporation is not responsible in any way for the editorial policy or other contents of the publication.

This publication is intended as a general guide. It covers a highly technical and complex subject and should not be used for making decisions concerning specific products or applications. This publication is sold as is, without warranty of any kind, either express or implied, respecting the contents of this publication, including but not limited to implied warranties for the publication, performance, quality, merchantability, or fitness for any particular purpose. Pinnacle Publishing, Inc., shall not be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this publication. Articles published in *FoxTalk* reflect the views of their authors; they may or may not reflect the view of Pinnacle Publishing, Inc. Inclusion of advertising inserts does not constitute an endorsement by Pinnacle Publishing, Inc. or *FoxTalk*.

The Subscriber Downloads portion of the *FoxTalk* Web site is available to paid subscribers only. To access the files, go to www.pinpub.com/foxtalk, click on "Subscriber Downloads," select the file(s) you want from this issue, and enter the user name and password at right when prompted.

User name `mellow`

Password `neutral`