COM Enhancements in VFP 7

Doug Hennig

This is last of a six-part series on language enhancements in VFP 7. This month's article focuses on those enhancements related to COM.

For the past six months, my column has focused on language enhancements in VFP 7. We started by discussing enhancements to existing commands and functions, then examined new ones. However, we concentrated on just general-purpose commands and functions, ignoring those related to an entire other area of enhancements in VFP 7: those providing COM-related enhancements. In this article, we'll look at the improvements in VFP 7 that make it better both as a COM server and a client of COM objects.

Some of the COM issues I'll discuss in this article may be new to you, especially if you haven't worked with COM much. Due to space limitations, I won't discuss the internal workings of COM in any detail. Instead, I suggest reading one of the many books available on this subject. In particular, I like the early chapters of Designing Component-Based Applications by Mary Kirtland, published by Microsoft Press (ISBN 0-7356-0523-8), because she doesn't assume you're an experienced C programmer and explains the concepts in easy-to-understand terms.

Session Enhancements

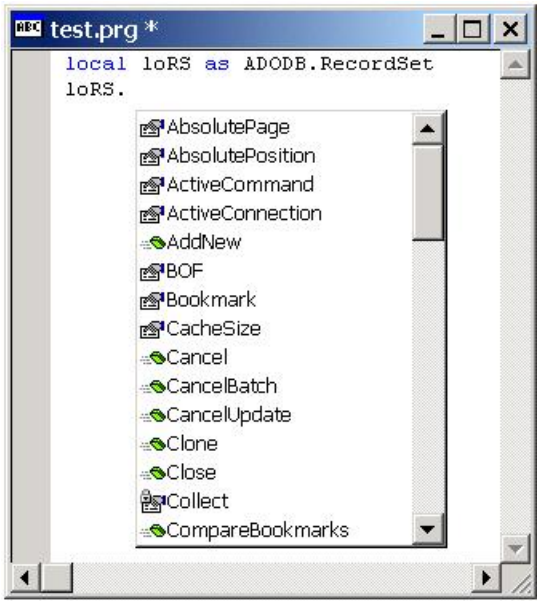
The first COM enhancements we'll discuss are to the Session class. In VFP 7, OLEPUBLIC subclasses of Session now expose only custom properties, events, and methods (PEMs) in the generated type library. This means you no longer have to explain to non-VFP users of your COM servers why they shouldn't worry about what BaseClass, ReadExpression, and ResetToDefault are for, nor do you have to create a subclass that specifically declares all base PEMs as protected. Another improvement is that the default settings for SET TALK, SET EXCLUSIVE, and SET SAFETY are now OFF, since those are the values you should use.

Strong Typing for Intellisense

One thing VFP developers have long envied about other interactive development environments (or IDEs) such as Visual Basic's is Intellisense for objects. For example, in an IDE that supports it, typing an object name then a period displays a dropdown list of the PEMs for that object. This is handy for a couple of reasons: you can choose a PEM from the list rather than having to type it (great for those long names like SelectedItemForeColor), which is not only faster but avoids spelling errors, and it allows you to see the PEMs for objects you may not be completely familiar with ("is that TreeView property called SelectedItem or SelectedNode?").

VFP 7 implements a very powerful (and extendible!) Intellisense. I won't go into details of Intellisense here, but here's how it relates to the COM enhancements we're discussing this month: you can tell VFP what type of object a variable will contain ("strong typing" the variable; see the next section for further discussion of strong vs. weak typing), and VFP will then provide Intellisense for that variable at design time. For example, Figure 1 shows what happened when I declared that loRS would contain a reference to an ActiveX Data Objects (ADO) RecordSet using the new AS clause of the LOCAL command (this clause is also supported for PUBLIC, PARAMETERS, and LPARAMETERS, but not PRIVATE), then typed "loRS" and hit the period key. Immediately, a listbox of the PEMs of a RecordSet object appeared. Icons in the list indicate which are properties (a document with a hand on it), events (a lightning bolt), and methods (what looks like a speeding eraser) and their scope (a lock indicates protected PEMs). Notice that I didn't actually instantiate a RecordSet in this code; that's not necessary for Intellisense to do its magic. Intellisense works on native as well as COM objects, and automatically works on scoped names or references (for example, it works for "This", "Thisform", "This.Parent", etc.).

Figure 1. Strong typing enables Intellisense for objects.



DEFINE CLASS and Type Library Enhancements

Classes defined non-visually (that is, in PRG files rather than the Class Designer) get a major facelift in VFP 7. Most of the improvements actually relate to the type library for a class, so they're really only applicable to OLEPUBLIC classes.

The first improvement, which isn't just for OLEPUBLIC classes, is in the DEFINE CLASS statement itself. You can now specify the location of the parent class for a subclass using the syntax DEFINE CLASS ClassName AS ParentClass OF ClassLibrary. This means you don't have to put all your classes in one big PRG nor do you have to worry about what SET CLASSLIB or SET PROCEDURE are set to. To see an example of this, run SUBCLASS.PRG (included with this month's Subscriber Downloads). It defines and instantiates a class called Class2, which is a subclass of Class1, which is defined in ParentClass.prg.

If you've ever used an object browser (such as the one in VBA-enabled applications like Word) to examine the type library for a VFP COM server, you've likely noticed that all VFP methods accept and return Variant values while non-VFP servers accept and return values of a certain type (such as Integer, String, etc.). The reason for this is that VFP is a weakly typed language; you can initialize a variable to a numeric value, then later store a string in it. While this hasn't been changed in VFP 7, what has changed is the ability to tell the type library the data type for method parameters and return values using the AS clauses for both. Here's an example:

```
function MyMethod(Param1 as String, Param2 as Integer) ;
  as Boolean
```

MyMethod accepts string and integer parameters and returns a boolean (logical) value. If this is a method of an OLEPUBLIC class, you'll see these data types listed in the type library when you view it in an object browser. Since VFP 7 isn't strongly typed, type enforcement is up to the compiler or language of the client; for example, these types aren't enforced when calling this method from a VFP client, whether through COM or not. You can also indicate if a parameter is passed by reference by placing an @ after the data type for the parameter. For methods that shouldn't return anything to the client, use AS VOID for the method return type.

In addition to data types, you can also specify a help string for a method using the new HELPSTRING clause. The string following this keyword will appear in the type library as helpful information for the method.

You may be wondering how you assign data types and help strings to properties. The new COMATTRIB feature provides those capabilities and more. COMATTRIB is an array you can define for each PEM using the syntax PEMName_COMATTRIB. The elements of this array contain the status of the PEM (such as hidden or read-only), a help string for a property, the name of the PEM as it should appear in

the type library (normally, VFP writes PEM names to the type library in all lowercase, so use this element to preserve capitalization), and the data type for a property.

TESTSERVER.PRG shows off these new features. Here's the code:

```
#define COMATTRIB_RESTRICTED   0x1
#define COMATTRIB_HIDDEN       0x40
#define COMATTRIB_NONBROWSABLE 0x400
#define COMATTRIB_READONLY     0x100000
#define COMATTRIB_WRITEONLY    0x200000

define class TestServer as Session olepublic

* Define a read-only ErrorMessage property.

  ErrorMessage = ''
  dimension ErrorMessage_COMAttrib[4]
  ErrorMessage_COMAttrib[1] = COMATTRIB_READONLY
  ErrorMessage_COMAttrib[2] = 'Contains details ' + ;
    'about errors'
  ErrorMessage_COMAttrib[3] = 'ErrorMessage'
  ErrorMessage_COMAttrib[4] = 'String'

* Define the COMATTRIB array for NameLength only so the
* proper capitalization of the method name is used.

  dimension NameLength_COMATTRIB[4]
  NameLength_COMAttrib[1] = 0
  NameLength_COMAttrib[2] = ''
  NameLength_COMAttrib[3] = 'NameLength'
  NameLength_COMAttrib[4] = ''

* Define a method to return the length of the passed
* string, using typed parameters and return value and a
* help string.

  function NameLength(Name as String) as Integer ;
    helpstring 'Returns the length of the specified name'
    return len(Name)
  endfunc
enddefine
```

To see how these type library improvements work, do the following:

- Create a DLL from the TEST project (which includes TESTSERVER.PRG) by typing BUILD DLL TEST.DLL FROM TEST.PJX in the Command window.
- Create a new Microsoft Word document and press Alt-F8 to display the Macros dialog. Type "test" for the macro name and click on the Create button to display the VBA editor.
- From the Tools menu, choose References, check the box in front of the "test Type Library" entry, and choose OK.
- Press F2 to display the Object Browser. In the upper combobox, choose "test", then select TestServer in the Classes pane.
- Notice the following. First, you don't see any of the base PEMs for the Session class (such as BaseClass), only the one property and one method we defined for this class. Next, the capitalization for the PEMs has been preserved. When you click on "ErrorMessage", you'll see the String data type specified for the property, the fact that it's read-only, and the help string (they're displayed in the bottom pane of the Object Browser). Click on "NameLength" to see the attributes of that method.
- Close the Object Browser, then type the following in the Code window:

```
Dim loTest As TestServer
Set loTest = New TestServer
loTest.ErrorMessage = "Howdy"
MsgBox (loTest.NameLength("Howdy"))
```

- Click on the first line of the code, then press F5 to run it. You'll get a compile error on the assignment to ErrorMessage because it's flagged as read-only. Comment out this line by placing a single quote at the start (VB's equivalent to an asterisk), then press F5 again. This time, you should see "5" appear in a messagebox, since that's the length of "Howdy".

PRG vs VCX

I mentioned above that PRG-based classes get a major facelift in VFP 7. The same is not true for VCX-based classes. That means we're going to have to get used to doing more class work in PRGs starting in VFP 7, especially for VFP COM servers. While this may seem like a big disadvantage over a visual class designer, there are some significant advantages to defining classes in PRGs, including:

- PRGs are simply text files while VCXs are tables. As a result, PRGs are very difficult to corrupt while VCXs are (being charitable) somewhat easier to corrupt.
- You can open and even modify the PRG for a class while the class is instantiated because it's the FXP for the PRG that's in use.
- Class maintenance is easier in a PRG. For example, while you can change the parent class of a VCX-based class using the Class Browser (or even opening the VCX as a table and changing the Classloc column), it's much easier to do that in a PRG. Also, moving code from a parent class to a subclass or vice versa is easy in PRGs since you can have both open at the same time; the Class Designer prevents you from simultaneously opening a VCX-based class and the parent of that class.

Of course, there are drawbacks to using PRGs too, including the lack of a Property Sheet. The Fox Wiki has a more complete discussion of the pros and cons of defining classes in PRGs available at http://fox.wikis.com/wc.dll?Wiki~PRGvsVCX.

Implementing and Supporting Interfaces

In VFP, we're used to creating a class that both defines the PEMs of an object and contains the code that makes up its behavior. In the COM world, these two things are usually distinct. The structure of a class (its PEMs) is called the interface and its behavior (the code) is the implementation. In VFP, we use subclassing to create a new class with the same PEMs as an existing one. Subclassing doesn't exist in COM. Instead, a class implements an interface; that is, it "inherits" the PEMs of an interface (but not the code, since there isn't any in an interface).

An example of this is an ADO RecordSet object. It implements two interfaces: _RecordSet, which has all the properties and methods you see in a RecordSet object, and RecordsetEvents, which defines the events a RecordSet object will trigger under certain conditions or when certain actions are taken. How did I know it implements these interfaces? I used the VFP 7 Object Browser, available from the Tools menu (see Figure 2). This cool new tool, which I won't go into detail about here, displays the classes, constants, enumerated values ("enums"), PEMs, and interfaces defined in COM libraries. I opened the "Microsoft ActiveX Data Objects 2.6 Library" (which shows as its library name ADODB in the Object Browser), turned on the "List interfaces in class details" item in the Options dialog, then selected the Recordset class under the Classes node and expanded the Interfaces node in the details pane.

Figure 2. The new VFP 7 Object Browser.

OK, what does all this have to do with VFP? Well, it turns out that the ability to implement interfaces is very useful or even crucial for at least a couple of things: receiving notification of events from COM objects and supporting COM+ features such as publishing and subscribing to COM+ events. We won't discuss COM+ here but we will look at event handling.

The DEFINE CLASS command has a new IMPLEMENTS keyword that allows the class to inherit the interface of a COM component. You can specify the interface to implement by providing the file name for the type library, the type library GUID (Globally Unique Identifier) and version, or the ProgID of the component. Here's an example, taken from ADOEVENTS.PRG included with this month's Subscriber Downloads:

```
DEFINE CLASS ADOEventHandler AS session OLEPUBLIC
  IMPLEMENTS RecordsetEvents IN ADODB.RecordSet
```

Instead of specifying ADODB.RecordSet, I could have used "c:\program files\common files\system\ado\msado15.dll" (the name of the file containing the type library) or "{00000205-0000-0010-8000-00AA006D2EA4}#2.5" (the type library GUID and version). The downside of these is that the first hard-codes the location of the type library (which may be in a different location on someone else's system) and the second is not exactly self-documenting and may take some work to find (I found this GUID by searching the Windows Registry for "RecordsetEvents", which was found in HKEY_CLASSES_ROOT\Interface\{00000266-0000-0010-8000-00AA006D2EA4}, and then taking the default value and the version value from the Typelib subkey).

A class implementing an interface must implement all the PEMs of that interface. To distinguish them from PEMs specific to the class, PEMs of the interface must be prefixed with the interface name (for example, the WillMove event of the RecordsetEvents interface is specified as RecordsetEvents_WillMove), and properties must be accessed through "get" (for reading) and "put" (for writing) methods (such as MyInterface_get_MyProperty). If an interface has a lot of PEMs, it can be a lot of work manually creating all these PEMs. Fortunately, there's an easier way: select the interface in the Object Browser and drag it to the code window for a PRG. For example, to create the ADOEventHandler class in ADOEVENTS.PRG, I typed MODIFY COMMAND ADOEVENTS.PRG, dragged the RecordsetEvents interface from the Object Browser to the ADOEVENTS.PRG window to create the definition of a class called MyClass that implements this interface, and then changed the class name to ADOEventHandler and filled in the code for a few methods. Here's the WillMove event, for example:

```
PROCEDURE RecordsetEvents_WillMove(adReason AS VARIANT, ;
  adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* start of my code
  lcMessage = 'WillMove event:' + chr(13) + 'Reason: '
  do case
    case adReason = ADRSNMOVE
      lcMessage = lcMessage + 'move'
    case adReason = ADRSNMOVEFIRST
      lcMessage = lcMessage + 'move first'
    case adReason = ADRSNMOVENEXT
      lcMessage = lcMessage + 'move next'
    case adReason = ADRSNMOVELAST
      lcMessage = lcMessage + 'move last'
  endcase
  wait window lcMessage
* end of my code
ENDPROC
```

Notice that the parameters were automatically listed and the data types of the parameters and method were automatically specified. You definitely will want to use the Object Browser to create classes this way!

Now that we've implemented an interface, let's see how we can use it to receive notification of events in a COM object. Visual Basic supports the "WITH EVENTS" clause of a class definition, meaning that it will automatically receive notification of the events fired by the interface it implements. VFP doesn't have this clause; instead, we have to use the new EVENTHANDLER() function to bind a COM object to a VFP class that will receive notification of events from the object. This function accepts three parameters: a reference to the COM object, a reference to the VFP object, and a logical parameter (.T. to unbind the objects, .F. or not passed to bind them). Here's an example, also taken from ADOEVENTS.PRG, that instantiates an ADO RecordSet object, instantiates the ADOEventHandler class described above, and binds the two together:

```
oRS     = createobject('ADODB.RecordSet')
oEvents = createobject('ADOEventHandler')
eventhandler(oRS, oEvents)
```

Now, any events raised by the RecordSet object will call the appropriate method in the ADOEventHandler object. For example, as you'll see when you run ADOEVENTS.PRG, moving the record pointer in the RecordSet with the MoveFirst, MoveNext, and MoveLast methods causes the RecordsetEvents_WillMove (before the move) and RecordsetEvents_MoveComplete (after the move) methods of ADOEventHandler to execute. This is a very powerful mechanism that, like putting code into the various methods of a VFP object, allows you to control what happens when an event is fired.

There are a couple of other enhancements in VFP 7 related to interfaces. GETINTERFACE() is a new function that returns an early-bound object reference to the specified interface for an existing COM object. CREATEOBJECTEX() can accept a new third parameter: the GUID for the interface (the interface ID or IID) of the COM server being instantiated. VFP will use the default interface of the server if you pass an empty string for IID. Since VFP communicates to COM servers through the IDispatch interface, it normally can't instantiate servers that don't support this interface; passing the IID allows VFP to work with these types of servers.

Other COM-related Enhancements

The remaining COM-related enhancements are described in the VFP help file as being for "advanced developers" who need these features. So, the majority of us mortals won't be worrying too much about them.

COMARRAY(), which determines how VFP passes arrays to COM objects, supports a new value for the second parameter that declares arrays as being fixed in size, meaning that COM servers can't resize the array. COMCLASSINFO(), which obtains information about a COM object, also supports a new value for the second parameter; passing 5 means you want to know the type of object it is: a native VFP object, an ActiveX control, a COM component, or an OLEBound object bound to a General field. COMPROP() is a new function that specifies how Unicode conversion and property assignment are handled for the specified property of a COM object. SYS(2336) provides critical section support in VFP COM servers. SYS(2340)

specifies whether VFP COM servers stay running or shut down when the current user logs out of Windows. SYS(3095) returns an outgoing IDispatch pointer for the specified COM object and SYS(3096) returns a reference to the COM object for the specified IDispatch pointer. SYS(3097) calls the IDispatch AddRef function for a COM object (which increments the reference counter for the object) and SYS(3098) calls the IDispatch Release function for a COM object.

Conclusion

This concludes our coverage of language enhancements in VFP 7. However, don't think that's all there is to new features in VFP 7. There's an almost overwhelming list of new things I haven't covered, including tons of IDE enhancements (Intellisense, new window behaviors including docking, editor enhancements such as better find features and bookmarks, etc.), database container events, new menu features (pictures, Office 2000-like appearance, visual support for "most recent used" functions, etc.), OOP improvements (MouseEnter and MouseLeave events, support for "hot tracking", multirow grid column headers, etc.), an OLE DB provider for VFP, more FoxPro Foundation Class (FFC) classes, and lots more. However, since the focus of this column is code and reusable tools, next month we'll get back to our regularly scheduled program.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's award-winning Stonefield Database Toolkit. 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 Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.