VFP: Ideal for Tools, Part 2

VFP provides lots of commands and functions for exploring classes and forms that help in building developer tools.

Tamar E. Granor, Ph.D.

In my last article, I wrote about VFP language elements related to data that are useful for building developer tools. This time, I'll focus on language related to classes and forms.

The VFP language makes it easy to write tools that enhance development. There are lots of functions and some commands that let you explore and modify the code you're working on. For class libraries and forms, there's a particularly rich set of ways to look inside and make changes.

Note that this article looks only at VCX-based classes. The next article in this series will show ways to peek inside PRG-based classes.

As in my last article, the examples in this article are mostly drawn from tools that come with VFP or are available in VFPX.

Treating VFP files as data

One of the things that has long made VFP's architecture so open is that many of the files used to store code are actually VFP tables. For example, the Form Designer creates a SCX/SCT pair. The SCX is really a DBF (table), while the SCT is an FPT (memo file). Table 1 shows the various VFP file types that are actually tables.

Table 1. VFP stores a variety of objects in tables.

File type	Table (DBF) extension	Memo (FPT) extension
Class library	VCX	VCT
Form	SCX	SCT
Label	LBX	LBT
Menu	MNX	MNT
Project	PJX	PJT
Report	FRX	FRT

Having all these definitions in tables means that you can use the normal data-handling tools of VFP to work on them. For example, the method in Listing 1 comes from the VFPX PEM Editor tool (specifically, the pemeditor_idex class); it checks whether a specified class is in a specified class library.

Listing 1. This method, called GoToDefProcessVCXForClass, comes from PEM Editor. It determines whether a particular class is defined in a particular class library.

```
Lparameters tcVCX, tcName, toInclude

*** Doug Hennig 2010-11-18
Local llOK, lnSelect
lnSelect = Select()
Select 0

Try
  Use (tcVCX) Again Shared Alias This_VCX
  llOK = .T.
Catch
  llOK = .F.
Endtry

If llOK
  Locate For (Lower (objname)) == ;
    Lower (tcName) ;
    And Lower(reserved1) = 'class' ;
    And Not Deleted()
  If Found()
    toInclude.File       = tcVCX
  EndIf
  Use
Endif

Select (lnSelect)
Return
```

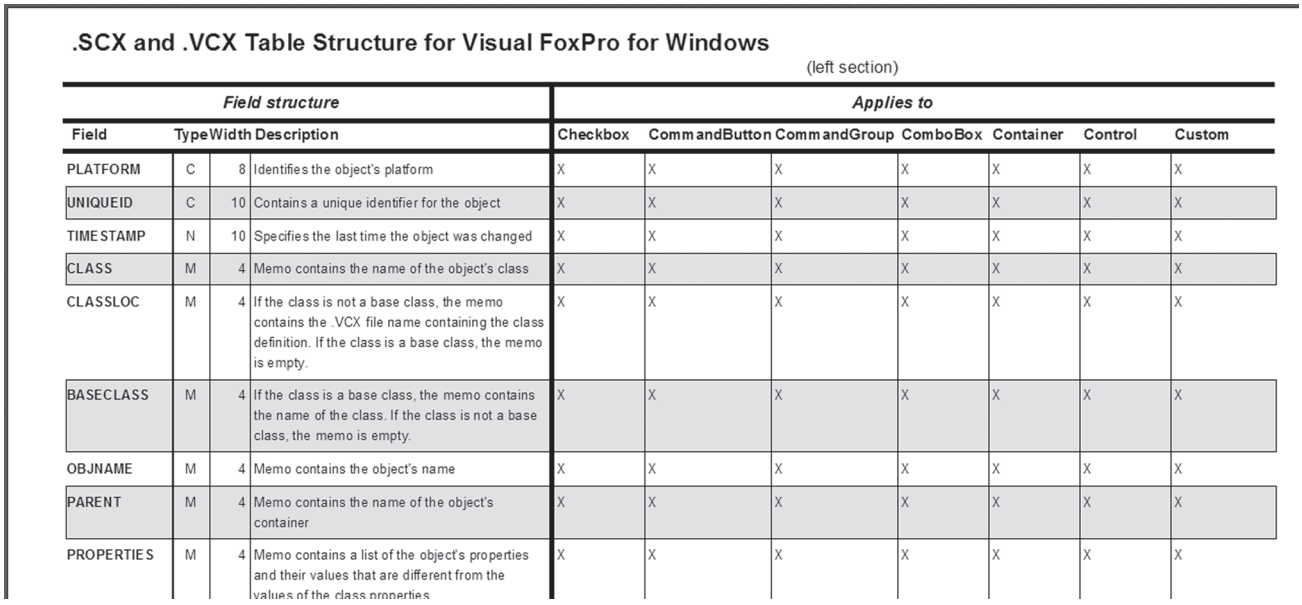
I've used the ability to treat VFP's forms, class libraries and projects as tables extensively in writing "quick and dirty" tools to handle a particular problem.

You can find documentation for many of these tables in the Tools\FileSpec folder of your VPF installation. It contains data and reports that document the fields. For example, Figure 1 shows part of report60scx1, which documents the SCX and VCX file structures.

Looking into class libraries

Two functions let you look inside class libraries to see what they contain. AVCXClasses() fills an array with information about the classes in a class library, while AGetClass() lets a user choose a class from a class library.

AVCXClasses() takes two parameters, an array and the name of a class library and fills the array with information about each class in the class library. The resulting array has eleven columns, described in Table 2.



.SCX and .VCX Table Structure for Visual FoxPro for Windows

(left section)

Field structure				Applies to						
Field	Type	Width	Description	Checkbox	CommandButton	CommandGroup	ComboBox	Container	Control	Custom
PLATFORM	C	8	Identifies the object's platform	X	X	X	X	X	X	X
UNIQUEID	C	10	Contains a unique identifier for the object	X	X	X	X	X	X	X
TIMESTAMP	N	10	Specifies the last time the object was changed	X	X	X	X	X	X	X
CLASS	M	4	Memo contains the name of the object's class	X	X	X	X	X	X	X
CLASSLOC	M	4	If the class is not a base class, the memo contains the .VCX file name containing the class definition. If the class is a base class, the memo is empty.	X	X	X	X	X	X	X
BASECLASS	M	4	If the class is a base class, the memo contains the name of the class. If the class is not a base class, the memo is empty.	X	X	X	X	X	X	X
OBJNAME	M	4	Memo contains the object's name	X	X	X	X	X	X	X
PARENT	M	4	Memo contains the name of the object's container	X	X	X	X	X	X	X
PROPERTIES	M	4	Memo contains a list of the object's properties and their values that are different from the values of the class properties	X	X	X	X	X	X	X

Figure 1. This is one of a group of reports that contain information about the structure of the various tables used to store VFP components.

Table 2. Each column of the array created by AVCXClasses() contains one information item about a class in the specified class library.

Column	Contains
1	Name of the class.
2	Base class.
3	Parent class.
4	Relative path and file name for the class library of the parent class. Empty if the parent class is a VFP base class.
5	Relative path and file name for the image specified as the class's custom icon. This is the Toolbar icon in the Class Info dialog.
6	Relative path and file name for the image specified as the class's icon for use in the Project Manager and Class Browser. This is the Container icon in the Class Info dialog.
7	Scale mode for the class, either "Pixels" or "Foxels."
8	Description of the class (from the Description editbox in the Class Info dialog).
9	Relative path and file name for the include file specified for the class. Empty if no include file is specified.
10	User-defined information for the class, from the User memo field in the VCX.
11	OLEPUBLIC status of the class, either .T. or .F.

The Toolbox uses AVCXClasses() when you add a class library to a category. The code in Listing 2 is drawn from the CreateToolsFromVCX method of the ToolboxEngine class. (Note that I've removed some of the code for brevity.) This code gets the list of classes in the specified class library, checks whether they're already in the Toolbox, and if not, loops through them, creating a tool for each.

Listing 2. The ToolboxEngine class's CreateToolsFromVCX method uses AVCXClasses() to find out what classes to add.

```
TRY
  m.nCnt = AVCXCLASSES(aVCXInfo, m.cFilename)
  *** Code to set up progress bar omitted

  FOR m.i = 1 TO m.nCnt
    *** Code to update progress bar omitted

    m.lDupe = .F.
    SELECT ToolboxCursor
    SCAN ALL FOR ParentID == m.cCategoryID ;
         AND SetID == m.cSetID
      m.oToolItem = THIS.GetToolObject( ;
        ToolboxCursor.UniqueID)
      IF VARTYPE(m.oToolItem) == 'O'
        * it's a duplicate, so ignore
        m.cClassName = LOWER(THIS.EvalText( ;
          NVL(oToolItem.GetDataValue( ;
          "classname"), '')))
        IF m.cClassName == ;
           LOWER(aVCXInfo[m.i, 1])
          m.lDupe = .T.
          EXIT
        ENDIF
      ENDIF
    ENDSCAN

    IF !m.lDupe
      m.cImageFile = THIS.GetImageForClass( ;
        aVCXInfo[m.i, 2], aVCXInfo[m.i, 5])

      m.cToolTip = aVCXInfo[m.i, 8]
        && class description
      m.oToolItem = THIS.CreateToolItem( ;
        m.cCategoryID, ;
        THIS.GenerateToolName( ;
        JUSTSTEM(m.cFilename), ;
        aVCXInfo[m.i, 1]), ;
```

```
        m.cToolTip, "CLASS", m.cImageFile, ;
        m.cSetID, '')

      IF VARTYPE(m.oToolItem) == 'O'
        oToolItem.SetDataValue("classlib", ;
          m.cClassLib)
        oToolItem.SetDataValue("classname", ;
          aVCXInfo[m.i, 1])
        oToolItem.SetDataValue("objectname", ;
          aVCXInfo[m.i, 1])
        oToolItem.SetDataValue("parentclass",;
          aVCXInfo[m.i, 3])
        oToolItem.SetDataValue("baseclass", ;
          aVCXInfo[m.i, 2])

        m.lUpdated = THIS.SaveToolItem( ;
          m.oToolItem, .T.)
      ENDIF
    ENDIF
  ENDFOR
```

AGetClass() is useful for the user interface of developer tools. It displays the Open dialog, set up to select a class library, and puts information about the selected class library into a two-element array. You can pass a number of parameters to customize the dialog; Listing 3 shows the syntax.

Listing 3. Use AGetClass() to have the user choose a class library.

```
lSuccess = AGetClass( ArrayName
             [, cClassLib
             [, cClass
             [, cDialogCaption
             [, cFileNameCaption
             [, cButtonCaption ] ] ] ] ] )
```

The cClassLib and cClass parameters specify a class library and class to highlight when the dialog opens. cDialogCaption provides a caption for the title bar; if you omit it, the caption is "Open." cFileNameCaption is the text to display instead of the default "File name:" next to the textbox that holds the name of the selected file. (It's the same as the second parameter to GetFile().) cButtonCaption specifies the text to appear on the OK button. (It's the same as the third parameter to GetFile().)

The Types page of the IntelliSense Manager calls AGetClass() in the Click method of the Classes… button, as shown in Figure 2. The relevant code is shown in Listing 4, though most of the method is omitted here.

Listing 4. This code is in the Click method of the Classes… button of the IntelliSense Manager.

```
IF aGetClass(aMyClass)
  lcFile = aMyClass[1]
  lcClass = aMyClass[2]
  IF !FILE(lcFile) OR EMPTY(lcClass)
    MESSAGEBOX(BADCLASSFILE_LOC,48)
    RETURN
  ENDIF
```

Getting a handle on a class or form

One of the unusual strengths of VFP is the ability to modify classes and forms programmatically at design-time, as well as at runtime. When the Form Designer or Class Designer is open, you can get a reference to the object being designed, as well as to the objects it contains. You can not only check their properties, events and methods (PEMs), but change them.

One way to get access to objects is ASelObj(). It fills an array with references to the selected objects in the Form Designer or Class Designer. The syntax for ASelObj() is shown in Listing 5, while Table 3 shows the possible values for the second parameter.

Listing 5. ASelObj() lets you grab references to whatever is selected in the Form or Class Designer.

```
nSelected = ASelObj( ArrayName [, nContainer])
```

ASelObj() is used widely in VFP tools. Listing 6 shows a fairly generic use, getting a reference to the selected object, or if no object is selected, the current form or class; it's drawn from the SetupEngine method of the MemberDataEngine class, the driver for the MemberData Editor.

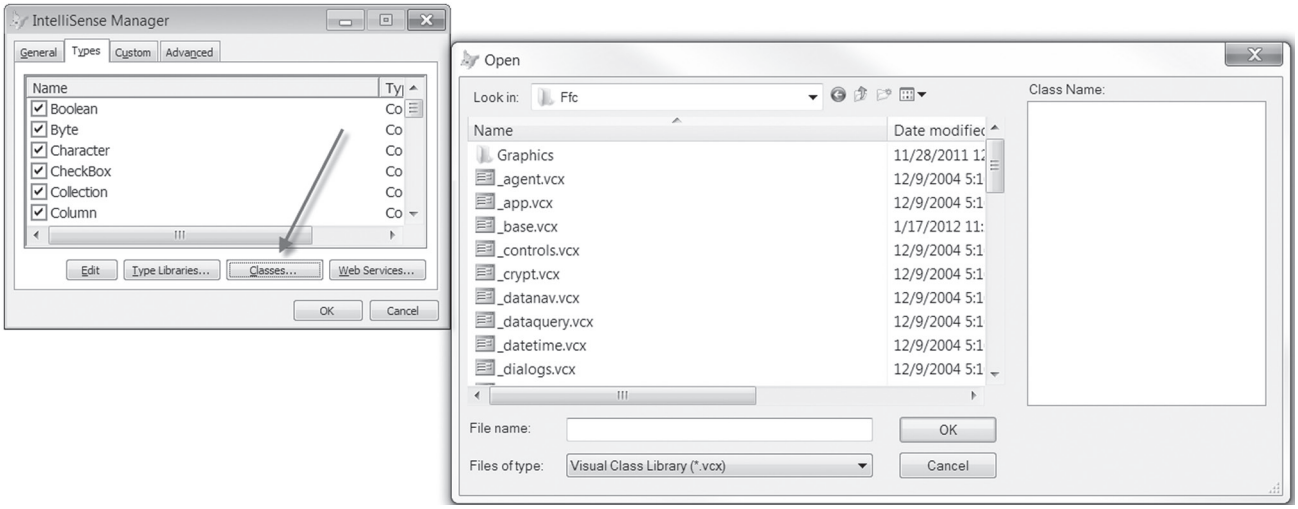


Figure 2. The Classes… button in the IntelliSense Manager calls AGetClass() to show the Open dialog, configured for choosing a class.

Listing 6. This code, from the SetupEngine method of the MemberDataEngine class, looks for one or more selected objects. If none are found, it gets a reference to the form or class being edited.

```
lnObjects = aselobj(laObjects)
if lnObjects = 0
  lnObjects = aselobj(laObjects, 1)
endif lnObjects = 0
if lnObjects > 0
  .oObject = laObjects[1]
endif lnObjects > 0
```

Table 3. The second parameter for ASelObj() determines what the function puts into the array.

Parameter value	Array contains
Omitted	One element for each selected object, with an object reference.
1	One element, an object reference to the container for the selected objects.
2	One element, an object reference to the data environment for the form. If the selected object is a class, the array is unchanged.
3	Three elements: An object reference to the container for the selected objects; The name of the SCX or VCX for the form or class; The path and name of the include file specified for the form or class; empty, if none is specified.

xTwo other functions, SYS(1270) and AMouseObj(), let you find out about the object under the mouse or, in the case of SYS(1270), at a specified location. These functions work both at design-time and at runtime. The syntax for these functions is shown in Listing 7.

Listing 7. The AMouseObj() and SYS(1270) functions both let you get a handle to an object at runtime or design-time.

```
nElements = AMouseObj( ArrayName
                       [, nRelativeToForm])
uReturn = SYS(1270 [, nXCoord, nYCoord ] )
```

AMouseObj() provides more data than SYS(1270). It puts four items in the array: an object reference to the object under the mouse, an object reference to that object's container, and the column and row (in pixels) of the mouse position. If you pass the optional second parameter, the second, third and fourth elements are based on the outermost container. In that case, at runtime or when the mouse is over a form being designed, the second element of the array is an object reference to the containing form; if the mouse is over the Class Designer, the second element is an object reference to the class being designed. With the nRelativeToForm parameter, the third and fourth elements of the array measure the mouse position relative to the object referenced in the second parameter.

The Toolbox uses AMouseObj() to prevent users from dropping items dragged out of the Toolbox onto the Toolbox itself (though you can drop onto a category header to add the item to that category). The code in Listing 8 appears in the DropObject method of the _root class from which all Toolbox tools are derived.

Listing 8. This code, from the _root.DropObject in the Toolbox code, prevents you from dropping an item dragged out of the Toolbox onto the Toolbox itself.

```
IF AMOUSEOBJ(aDropTarget, 1) > 0 AND ;
   LOWER(aDropTarget[2].Name) = "toolbox"
  RELEASE m.aDropTarget
  RETURN
ENDIF
```

While SYS(1270) provides only an object reference, it has flexibility that AMouseObj() doesn't. You can pass a point (that is, X and Y coordinates) and the function tells you what's under the specified point. There are two tricky issues here. First, the coordinates you pass are relative to the screen, not to the form you're running or even to VFP. So you may need to add _VFP.Left and _VFP.Top, respectively, to the coordinates of the point you're interested in to get the right answer. The second issue is that if you specify a point that isn't inside VFP, the function returns .F., so you need to check the return value's type before treating it as an object.

The Thor tool, Insert full name of object under mouse, uses SYS(1270) to find out what object it is. Listing 9 shows that part of the tool code.

Listing 9. This code in the Thor tool Insert full name of object under mouse finds the relevant object using SYS(1270). Later code in the tool (not shown here) finds the full name (including path) of the object referenced by bb.

```
bb    = Sys(1270)
If 'O' # Vartype (bb)
  Return
Endif
```

Examining and modifying objects

Once you have access to an object, you can look at or change its properties. But VFP also provides tools that let you determine the properties and methods of an object, and at design-time, examine and modify method code as well as properties.

What's in there?

There are several ways to find out what PEMs an object has. To get a complete list for the object, use the AMembers() function. To learn about a particular PEM, use PEMStatus().

AMembers() fills an array with information about the PEMs of a specified object. Exactly what information you get is determined by the param-

eters you pass. Listing 10 shows the syntax of the function, while Table 4 shows the possible values for nInfoType. As the table indicates, AMembers() can address COM objects as well as VFP objects, though there are some restrictions.

Listing 10. The AMembers() function fills an array with information about the object you pass.

```
nMembers = AMEMBERS( ArrayName, oObject
             [, nInfoType [, cFlags] ] )
```

Table 4. AMembers() can collect several different sets of information, depending on which value you pass for the third parameter.

nInfoType	Array contains
Omitted or 0	An alphabetical list of the object's properties. The array is one-dimensional.
1	A complete list of the object's PEMs and member objects. The array has two columns, with the names in the first and a string indicating the type of member in the second.
2	A list of the object's member objects. The array is one-dimensional.
3	A complete list of the object's PEMs with additional information about each. The array has either four or five columns, depending on whether the object is a VFP object or a COM object, and on the value of the cFlags parameter.

The fourth parameter lets you filter the PEMs included in the array. Pass a string containing one or more of the letters in Table 5 to include PEMs with one or more of the specific characteristics. By default, the results are unioned, so you get any PEMs that have any of the characteristics. If "+" is included in the parameter, the other items are intersected, so you get only PEMs that have all of the specified characteristics. Finally, include "#" in cFlags, and the resulting array has five columns, with the fifth containing the flag characters that apply to each PEM.

AMembers() is extremely valuable when building any tool that needs to address the list of PEMs for an object. Listing 11 shows code from the Object Inspector I built (see the January, 2011 issue of FoxRockX) that puts an object's properties and their values into a cursor, so they can be shown in a grid in the right pane. Figure 3 shows the tool in use, with an arrow indicating the grid.

Listing 11. This code from the Object Inspector put properties and their values into a cursor for display in a grid.

```
nPropCount = AMEMBERS(aProps, m.oObject, 0)
FOR nProp = 1 TO m.nPropCount
  cType = TYPE("oObject." + aProps[m.nProp])
  IF m.cType <> "U"
    cValue = TRANSFORM(EVALUATE("oObject." + ;
             aProps[m.nProp]))
  ELSE
    cValue = ;
      "<Property could not be evaluated>"
  ENDIF

  INSERT INTO (This.cCursorAlias) ;
    VALUES (aProps[m.nProp], m.cType, ;
            m.cValue)
ENDFOR
```

Table 5 shows the flag characters in groups to make it easier to understand the choices.

Table 5. AMembers()' fourth parameter lets you limit the PEMs that are contained in the array by specifying some or all of these flags.

Flag Character	Group	Indicates
"G"	Visibility	Include public PEMs.
"H"	Visibility	Include hidden PEMs.
"P"	Visibility	Include protected PEMs.
"N"	Origin	Include native PEMs, that is, those that are part of the object's base class.
"U"	Origin	Include user-defined PEMs, those added at some point in the class hierarchy.
"B"	Inheritance	Include PEMs defined at this level (that is, not inherited).
"I"	Inheritance	Include PEMs inherited from another class.
"C"	Changed	Include PEMs changed at some level of the class hierarchy.
"R"	Read-only	Include read-only PEMs.
"+"	Management	Combine the characters in cFlags with "and" rather than "or."
"#"	Management	Add a flags column to the array.

If you only need information about a single PEM, AMembers() is overkill. PEMStatus() provides much of the same information, one PEM at a time. Listing 12 shows the syntax for PEMStatus(), while Table 6 shows the legal values for nAttribute. As you can see, several of the attributes here map directly to flags for AMembers().

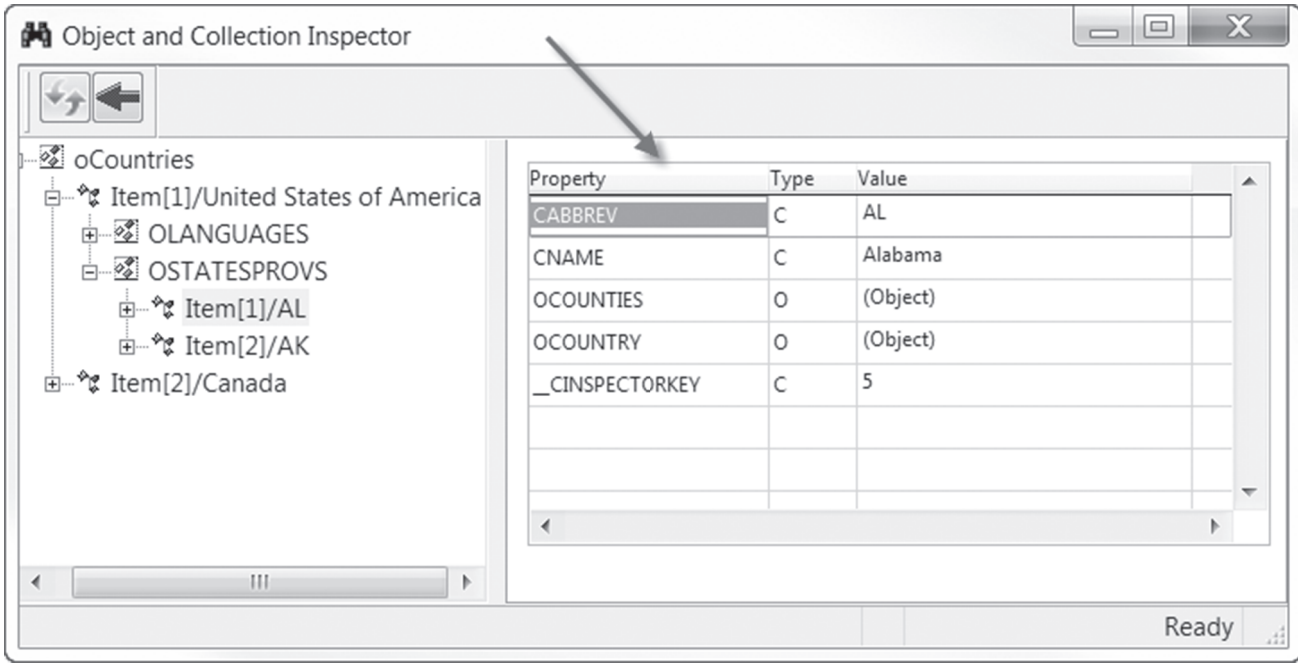


Figure 3. The code in Listing 11 is used to create the cursor that populates the grid in this tool.

Listing 12. PEMStatus() answers questions about a particular property, event or method.

```
uResult = PEMStatus( oObject, cPEMName,
                     nAttribute )
```

Table 6. The nAttribute parameter tells PEMStatus() what information to return.

nAttribute	PEMStatus() returns
0	A logical value that indicates whether the PEM has been changed.
1	A logical value that indicates whether the property is read-only. Applies only to properties.
2	A logical value that indicates whether the property is protected.
3	A string that indicates whether the specified PEM is a property, event, method or object.
4	A logical value that indicates whether the PEM is user-defined.
5	A logical value that indicates whether the object has the specified PEM.
6	A logical value that indicates whether the PEM was inherited from a class higher in the inheritance hierarchy.

I use nAttribute=5 more than anything else, so I can write generic code that checks whether a given property or method exists. But you can also use it for such things as ensuring that you can write a new value to a property. The code in Listing 13, drawn from the Toolbox (specifically, the OnCompleteDrag method of the _ImageTool class) does both.

Listing 13. This code from the Toolbox, checks whether the Left and Top properties are read-only or protected before attempting to change their values. For the Visible property, it checks those two, but first checks whether the object has that property.

```
IF VARTYPE(m.oObjRef) == 'O'
  IF !PEMSTATUS(m.oObjRef, "Left", 1) AND ;
     !PEMSTATUS(m.oObjRef, "Left", 2)
    m.oObjRef.Left = m.nMouseXpos
  ENDIF
  IF !PEMSTATUS(m.oObjRef, "Top", 1) AND ;
     !PEMSTATUS(m.oObjRef, "Top", 2)
    m.oObjRef.Top  = m.nMouseYPos
  ENDIF
  IF PEMSTATUS(m.oObjRef, "Visible", 5) AND ;
     !PEMSTATUS(m.oObjRef, "Visible", 1) AND ;
     !PEMSTATUS(m.oObjRef, "Visible", 2)
    m.oObjRef.Visible = .T.
  ENDIF
ENDIF
```

Making changes at design-time

For developer tools, we generally need the ability to change things at design-time. As the previous example demonstrates, it's easy to change the value of a property at design-time. But we also may want to change method code. In addition, sometimes what we want to store in a property is not a value, but an expression to be evaluated at runtime. A set of four methods gives us the ability to see the content of any PEM, and to change it.

ReadMethod and WriteMethod apply to method code. Use ReadMethod to retrieve the current code for any method of an object; the syntax is shown in Listing 14.

Listing 14. The ReadMethod method of each VFP base class (except Empty) lets you retrieve the code for any method.

```
cCode = oObject.ReadMethod(cMethod)
```

The Find capability of PEM Editor uses ReadMethod to figure out whether a method has code, as shown in Listing 15. (Note the use of PEMStatus() as well to ensure that the specified object actually has the method in question and has been changed.)

Listing 15. PEM Editor's Find capability uses ReadMethod to determine whether a method has code at this level.

```
Function NonDefault (lcName)
  If Pemstatus(goObject, lcName, 5) And ;
     Pemstatus(goObject, lcName, 0)
    If Inlist(Pemstatus(goObject, lcName, 3),;
              'Method', 'Event')

      Return Not Empty( ;
        goObject.ReadMethod(lcName))

    Else
      Return .T.
    Endif
  Else
    Return .F.
  Endif
EndFunc
```

The corresponding WriteMethod method lets you store code in a method. You can even add a method and give it code all at once. Listing 16 shows the syntax. The lAddMethod parameter indicates whether to add the method if it doesn't already exist. Use nVisibility to indicate whether your new method is public (1), private (2) or hidden (3). As you'd expect, the cDescription parameter lets you provide a description for the method that appears in the Property Sheet.

Listing 16. Use WriteMethod to create methods and to populate them.

```
oObject.WriteMethod(cMethodName, cMethodText
                    [, lAddMethod
                    [, nVisibility
                    [, cDescription ] ] ] )
```

The code in Listing 17 comes from the PEMEditor_Utils class library of the PEM Editor; it's used when copying PEMs from one object to another.

Listing 17. This code, from the DoPasteProperties method of PEMEditor_Utils, uses WriteMethod to create a new method, if necessary, and to store the copied code.

```
For lnRow = 1 To Alen( ;
   This.oServer.aCopiedProperties, 1)
  lcPem = ;
    This.oServer.aCopiedProperties(lnRow, 1)

  lcDescript = Rtrim(  ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccPasteDescriptCol))
  lcType     = ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccPasteTypeCol)
  lxValue    = ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccPasteValueCol)
  lbNew      = ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccPastelNewCol)
  lbSelect   = ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccPasteSelectCol)
  lnVisibility   = ;
    This.oServer.aCopiedProperties(lnRow, ;
    ccVisibilityCol)

  Try
    Do Case
      Case Not ;
       (This.oServer.aCopiedProperties(lnRow,;
       ccNonDefaultCol) Or llAll)

      Case Not lbSelect

      Case Upper (lcPem) == '_MEMBERDATA'

      Case lcType = 'M'
        If lbNew

          loObject.WriteMethod(lcPem, ;
            lxValue, .T., lnVisibility, ,
            lcDescript)

        Else
          loObject.WriteMethod(lcPem, lxValue)
            && , lnVisibility, lcDescript)
        Endif

      Case lcType = 'X'
        If lbNew
          loObject.AddProperty(lcPem, ;
            lxValue, lnVisibility, lcDescript)
        Endif
        loObject.WriteExpression(lcPem, ;
          lxValue, lnVisibility, lcDescript)

      Case lcType = 'V'
        If lbNew
          loObject.AddProperty(lcPem, ;
            lxValue, lnVisibility, lcDescript)
        Else
          loObject.WriteExpression(lcPem, '')
          loObject.AddProperty(lcPem, ;
            lxValue, lnVisibility, lcDescript)
        Endif

    Endcase
    **** Code related to memberdata removed
    **** for this example
    ****************************************
****

  Catch To loException
    lcErrors = lcErrors + lcPem + ": " + ;
      loException.Message + " (" + ;
      Transform(loException.ErrorNo) + ")" + ;
      CR
  Endtry
Endfor
```

You might wonder why you need methods to read and write properties since you can just access a property value directly. The ReadExpression and WriteExpression methods let you deal with properties where an expression is assigned rather than a value. For example, you might have a property called dToday to hold today's date; in the property sheet, it would be assigned "=DATE()." Checking

the value of dToday would give you the actual date, not the expression. But ReadExpression returns the expression.

The syntax for these methods is pretty simple; it's shown in Listing 18.

Listing 18. The ReadExpression and WriteExpression methods let you work with expressions stored in the Property Sheet.

```
cPropertyExpression =
  oObject.ReadExpression( cProperty )
oObject.WriteExpression( cProperty,
  cPropertyExpression )
```

PEM Editor uses both of these functions in a method that lets users enter property values or expressions. The Expression Builder method, shown in Listing 19, retrieves the current value of a property using ReadExpression, then calls the Expression Builder (using the GetExpr() function), then writes the result back to the property using WriteExpression.

Listing 19. This method from PEM Editor lets a user edit the value of a property.

```
Procedure ExpressionBuilder( loObject, lcPEM)
  Local lcExpression, lcNewExpression, ;
        loException

  lcExpression = Substr( ;
    loObject.ReadExpression(lcPEM), 2)

  Getexpr (lcPEM) To lcNewExpression ;
    Default (lcExpression)

  Try
    loObject.WriteExpression(lcPEM, ;
      lcNewExpression)
  Catch To loException

  Endtry

Endproc
```

Programs and Projects up next

The final installment of this series will explore the language elements VFP provides for working with programmatic code and projects.

Authoren Profil

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of nearly a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer *available at www.foxrockx.com Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*