Create Your Own Property Editors

Doug Hennig

VFP 9 makes it possible to create your own editors for custom properties of classes and forms. This month, Doug Hennig presents a framework for these editors, allowing you to focus on the editor itself rather than the plumbing necessary to hook in the editor.

Since its release, VFP has provided editors that make it easier to specify the value of a property in the Properties window. Some properties, such as BorderStyle, have a combobox of acceptable values you can choose from. Others have a button with an ellipsis that launches the appropriate dialog, such as the Color dialog for BackColor and ForeColor. For years, VFP developers have lobbied Microsoft for the ability to specify editors for our own custom properties. In VFP 9, they deliver.

You specify an editor for a property in the script attribute of the Member Data for the property. Member Data is something new in VFP 9; it's an XML string that defines various attributes about the members of an object, including what case a member should be displayed in and whether it appears on the Favorites tab of the Properties window. (I discussed Member Data in detail in my June 2004 FoxTalk column, "MemberData and Custom Property Editors.") Member Data is stored in one of two places: a custom _MemberData property of the object, which can contain Member Data for multiple members of the object, or in records in the IntelliSense table (FoxCode.DBF), each of which contains the Member Data for a single member.

Rather than typing XML, which is tedious, VFP 9 includes the MemberData Editor, accessible from the Form and Class menus. To specify the editor for a property, select that property in the MemberData Editor, turn on the Has MemberData setting, and type the code to execute when the editor for that property is invoked in the Script editbox.

One of the interesting things about Member Data is that it's extensible; you can add custom attributes to the XML. One of the uses we'll make of this is storing meta data for our property editors. The MemberData Editor makes it easy to add custom attributes to a member—on the User-Defined page (see **Figure 1**), click Add to add a new custom attribute and then enter the value for that attribute in the Value editbox.

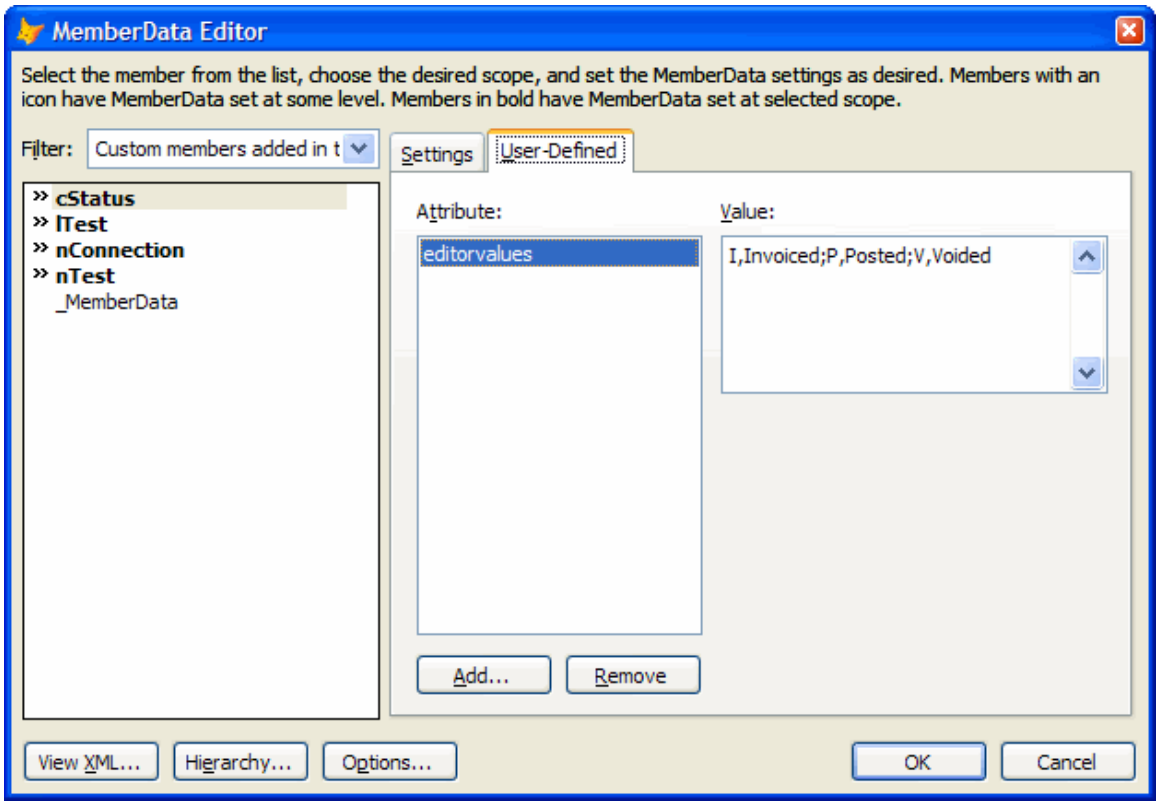

Figure 1. You can define custom Member Data attributes in the User-Defined page of the MemberData Editor.

Before we look at how to create property editors, here are a few things you should know about them:

- You can invoke the editor for a property in one of two ways: click on the button with the ellipsis that appears when an editor is specified for a property or double-click the property row in the Properties window.
- Property editors are hard to debug: if you put SET STEP ON into the code, execution is suspended at that point. However, even if you choose Step Into, execution continues without stopping. To prevent this, use SYS(2030, 1), which enables debugging in system components, just before the SET STEP ON.
- If you want to use classes in earlier versions of VFP, you can't use _MemberData. Well, you can, but _MemberData must have less than 255 characters or you won't be able to open the class in earlier versions, and you'll find it's very easy to surpass that limit given the verbosity of XML. In that case, store Member Data in the IntelliSense table instead.

PropertyEditor.PRG

Because much of the work in editing a property is the same from one editor to the next, I created a framework for property editors. The starting point is PropertyEditor.PRG; specify this program as the property editor in the Member Data for properties. PropertyEditor.PRG accepts as parameters the name of the editor and optionally the property the editor will edit.

PropertyEditor.PRG doesn't do very much; it simply instantiates and calls the appropriate class for the specified editor name. Editor classes are registered in PropertyEditors.DBF, a simple table in the same directory as the PRG with ID, NAME, CLASS, and LIBRARY columns.

Here's the Member Data for some of the properties in the Test class in Test.VCX. The lTest property, which contains a logical value, specifies the Toggle editor. (Note: you'll likely need to specify the path to PropertyEditor.PRG but that's been excluded in these samples for simplicity.)

```
<memberdata name="ltest"
type="property" display="lTest"
script="do PropertyEditor with 'Toggle', 'lTest'"/>
```

nTest contains numeric values between 1 and 3, so it also specifies Toggle but uses custom lowvalue and highvalue attributes to specify the range of values.

```
<memberdata name="ntest"
type="property" display="nTest"
script="do PropertyEditor with 'Toggle', 'nTest'"
lowvalue="1" highvalue="3"/>
```

cStatus specifies the status of an invoice: I means invoiced, P means posted, and V means voided. Since these are enumerated values, cStatus uses the Enumerated editor and its custom editorvalues attribute provides a list of the values in the format "value, description to display; value, description to display; etc."

```
<memberdata name="cstatus"
type="property" display="cStatus"
script="do PropertyEditor with 'Enumerated',
'cStatus'"
editorvalues="I,Invoiced;P,Posted;V,Voided"/>
```

PropertyEditor.PRG begins by opening PropertyEditors.DBF if necessary, looking for the specified name in the NAME column, and if found, instantiating the class specified in the CLASS and LIBRARY columns. Note that it uses TEXTMERGE() on LIBRARY, so you can specify a library in the VFP home directory using "HOME() + 'LibraryName.VCX'." Also, if a path isn't specified, PropertyEditor.PRG looks for the library in the same directory it's in. After instantiating the class, PropertyEditor.PRG calls the editor's EditProperty method to do the work.

```
lparameters tuEditorName, ;
  tcProperty
local lcDirectory, ;
  lcLibrary, ;
  loEditor

* Determine the directory we're running in.

lcDirectory = addbs(justpath(sys(16)))

* Create the specified editor, handling the cases
* where it isn't specified or isn't registered.

do case
  case not vartype(tuEditorName) $ 'CN' or ;
    empty(tuEditorName)
    messagebox('Must specify editor name.', 48, ;
      'Property Editor')
    return
  case not OpenEditorsTable(lcDirectory)
    return
  case (vartype(tuEditorName) = 'C' and ;
    seek(upper(tuEditorName), 'PropertyEditors', ;
    'Name')) or (vartype(tuEditorName) = 'N' and ;
    seek(tuEditorName, 'PropertyEditors', 'ID'))
    lcLibrary = textmerge(PropertyEditors.Library)
    if not file(lcLibrary)
      lcLibrary = forcepath(lcLibrary, lcDirectory)
    endif not file(lcLibrary)
    if file(lcLibrary)
      loEditor = newobject(PropertyEditors.Class, ;
        lcLibrary)
    else
      messagebox('The library specified for ' + ;
        transform(tuEditorName) + ;
        ' cannot be located.', 48, 'Property Editor')
```

```
      return
    endif file(lcLibrary)
  otherwise
    messagebox(transform(tuEditorName) + ;
      ' is not a registered editor.', 48, ;
      'Property Editor')
    return
endcase
use in PropertyEditors

* If a property name was specified (for example, for
* an editor that can work with several properties),
* set the cProperty property of the editor to it.

if vartype(tcProperty) = 'C' and not empty(tcProperty)
  loEditor.cProperty = tcProperty
endif vartype(tcProperty) = 'C' ...

* Tell the editor to do its thing.

loEditor.EditProperty()
```

SFPropertyEditor

SFPropertyEditor, defined in a VCX of the same name, is the base class for property editors. SFPropertyEditor shouldn't be used directly; subclass it to create the desired editor. Since PropertyEditor.PRG calls the EditProperty method of the editor, let's start there.

The first task is to get a reference to the object or objects selected in the Form or Class Designer. The GetObjectReference method fills and returns a collection with the references, so EditProperty puts the collection into the oObjects property. Next, EditProperty calls ValidateProperty to ensure that a property name was specified in the cProperty property and that the specified property is a member of the selected object. (PropertyEditor.PRG puts the second parameter it's passed into cProperty; for single property editors, fill in a value in the Properties window for the editor subclass.) GetCurrentValue returns the current value of the property (so the editor can display it if necessary) for the first selected object into the uCurrentValue property and cDataType contains the data type of that value. EditProperty then calls GetAttributes to read the Member Data attribute names and their values into the oAttributes collection so the editor can use them if necessary. If your editor needs to test that things are set up properly (for example, that cDataType contains the proper data type the editor is expecting), put that code into the TestCustomAsserts method that EditProperty calls; that method is abstract in this class. Finally, the PropertyEditor method, which is also abstract in this class, is called to perform the actual editing tasks.

```
local loException as Exception
with This
  try

* Get a reference to the selected objects.

    .oObjects = .GetObjectReference()

* Ensure we have a valid property name and get its
* current value and any attributes in _MemberData.

    .ValidateProperty()
    .uCurrentValue = .GetCurrentValue()
    .cDataType     = vartype(.uCurrentValue)
    .oAttributes   = .GetAttributes()

* Test any other asserts, then call the "real" editor
* method.

    .TestCustomAsserts()
    .PropertyEditor()

* Handle a failed assert.

  catch to loException ;
```

```
    when not empty(loException.UserValue)
    .Warning(loException.UserValue)

* Handle any other type of error.

  catch to loException
    .Warning(loException.Message)
  endtry
endwith
```

We won't look at all of the other methods in SFPropertyEditor, just those that contain important or interesting code.

GetObjectReference returns a collection of selected objects. It uses ASELOBJ() to fill an array of these objects and copies the references into a collection that it returns. Note the call to the Assert method; this method simply tests the specified condition and if it's false, THROWs the specified error message. Assert is similar to the ASSERT command but allows me to consolidate all assertion failure into one place: the first CATCH statement in EditProperty.

```
local laObjects[1], ;
  loObjects, ;
  lnI
This.Assert(aselobj(laObjects) > 0 or ;
  aselobj(laObjects, 1) > 0, ;
  'No object is selected.')
loObjects = createobject('Collection')
for lnI = 1 to alen(laObjects)
  loObjects.Add(laObjects[lnI])
next lnI
return loObjects
```

GetAttributes fills a collection with the names and values of all Member Data attributes defined for the property the editor is editing. It uses an XML DOMDocument object to make short work of the searching and parsing. Note that if the object doesn't have a _MemberData property or it's empty, it obviously must have "global" Member Data, which is stored in the IntelliSense table, or else the property editor wouldn't have been launched. So, the Member Data is read from the TIP memo field of the appropriate record in the IntelliSense table in that case.

```
local loAttributes, ;
  lnSelect, ;
  loDOM as MSXML2.DOMDocument, ;
  loObject, ;
  loNode as MSXML2.IXMLDOMElement, ;
  loAttribute as MSXML2.IXMLDOMAttribute, ;
  lcName, ;
  lcValue
loAttributes = createobject('Collection')
try

* Create an XML DOM object and load the XML in the
* _MemberData property of the first selected object.
* If that property doesn't exist or is empty, see if
* there's any global Member Data for it in the
* IntelliSense table.

  loDOM = createobject('MSXML2.DOMDocument.4.0')
  loDOM.async = .F.
  loObject = This.oObjects.Item(1)
  if pemstatus(loObject, '_MemberData', 5) and ;
    not empty(loObject._MemberData)
    loDOM.loadXML(loObject._MemberData)
  else
    select 0
    use (_foxcode) again shared alias __FOXCODE
    locate for TYPE = 'E' and ;
      upper(ABBREV) = upper(This.cProperty)
    if found()
```

```
      loDOM.loadXML(TIP)
    endif found()
    use
  endif pemstatus(loObject, '_MemberData', 5) ...

* Find the node for the property we're editing. If
* there is one, load all of the attributes into the
* collection.

  loNode = loDOM.selectSingleNode('//memberdata' + ;
    '[@name="' + lower(This.cProperty) + '"]')
  if vartype(loNode) = 'O'
    for each loAttribute in loNode.attributes
      lcName  = loAttribute.name
      lcValue = loAttribute.value
      loAttributes.Add(lcValue, lcName)
    next loAttribute
  endif vartype(loNode) = 'O'
catch
  use in select('__FOXCODE')
endtry
select (lnSelect)
return loAttributes
```

UpdateProperty isn't called from EditProperty but can be called from PropertyEditor in your subclass to write the new value to the property in each of the selected objects.

```
lparameters tuValue
local loObject, ;
  lcProperty
for each loObject in This.oObjects
  lcProperty = 'loObject.' + This.cProperty
  store tuValue to (lcProperty)
next loObject
```

SFPropertyEditorToggle

Let's look at a practical example. While VFP 9 was still in beta, I watched VFP guru Rick Schummer double-click a property in the Properties window to change it from .F. to .T. No big deal, except I did a double-take when I realized it was a custom rather than native property he'd double-clicked. I asked him whether that was some new feature that'd escaped my attention, and he gave a sly grin and said, "Nope, it's a custom property editor." After thinking about it a bit, I realized that such an editor would also be useful for numeric properties so they'd work just like double-clicking native properties with a pre-defined range of values such as BorderStyle. So I created SFPropertyEditorToggle.

The PropertyEditor method checks to see if custom lowvalue and highvalue Member Data attributes exist, and if so, that both of them exist and they have reasonable values (feel free to change this range check if you wish). It then toggles the property by calling UpdateProperty with the appropriate value: NOT the current value for a logical property, the current value + 1 for a numeric property that hasn't reached the maximum value yet, or the minimum value if the property is at the maximum.

```
local lnValue, ;
  lnLowValue, ;
  lcLowValueType, ;
  lnHighValue, ;
  lcHighValueType
with This

* Ensure that if either lowvalue or highvalue
* attributes exist that they both exist and that
* they're reasonable values.

  if .oAttributes.GetKey('lowvalue') > 0
    lnValue    = .oAttributes.Item('lowvalue')
    lnLowValue = int(val(lnValue))
  endif .oAttributes.GetKey('lowvalue') > 0
  lcLowValueType = vartype(lnLowValue)
```

```
  if .oAttributes.GetKey('highvalue') > 0
    lnValue    = .oAttributes.Item('highvalue')
    lnHighValue = int(val(lnValue))
  endif .oAttributes.GetKey('highvalue') > 0
  lcHighValueType = vartype(lnHighValue)
  .Assert(lcLowValueType = lcHighValueType, ;
    '_MemberData must specify both lowvalue and ' + ;
    'highvalue attributes.')
  .Assert(lcLowValueType = 'L' or ;
    (lcLowValueType = 'N' and ;
    between(lnLowValue, 0, 10)), ;
    'The _MemberData lowvalue attribute must be ' + ;
    '0 - 10.')
  .Assert(lcHighValueType = 'L' or ;
    (lcHighValueType = 'N' and ;
    between(lnHighValue, 0, 10)), ;
    'The _MemberData highvalue attribute must ' + ;
    'be 0 - 10.')

* Toggle the property value.

  do case
    case .cDataType = 'L'
      .UpdateProperty(not .uCurrentValue)
    case lcHighValueType = 'N' and ;
      .uCurrentValue >= lnHighValue
      .UpdateProperty(lnLowValue)
    otherwise
      .UpdateProperty(.uCurrentValue + 1)
  endcase
endwith
```

To see this editor in action, open the Test class in Test.VCX and double-click the lTest or nTest properties. As we saw earlier, the Member Data for both properties specifies Toggle as the editor name passed to PropertyEditor.PRG, and Toggle is registered in PropertyEditors.DBF as the SFPropertyEditorToggle class. Also, nTest has lowvalue and highvalue attributes that specify the range of values for nTest as 1 to 3. Double-click nTest several times to see the complete range of values.

SFPropertyEditorEnumerated

Some properties contain "code" values, such as BorderStyle, in which 0 means no border, 1 means a fixed single border, 2 means a fixed dialog border, and 3 means a resizable border. Since there's a pre-defined list of values and what they mean, these are enumerated properties. You may have your own enumerated properties, such as an invoice status property that contains I for invoiced, P for posted, or V for voided. Rather than forcing the developer to remember the possible values, use SFPropertyEditorEnumerated as the editor for these properties. As you can see in **Figure 2**, it displays a list of the values and their descriptions, similar to the combobox VFP displays in the Properties window for native enumerated properties.

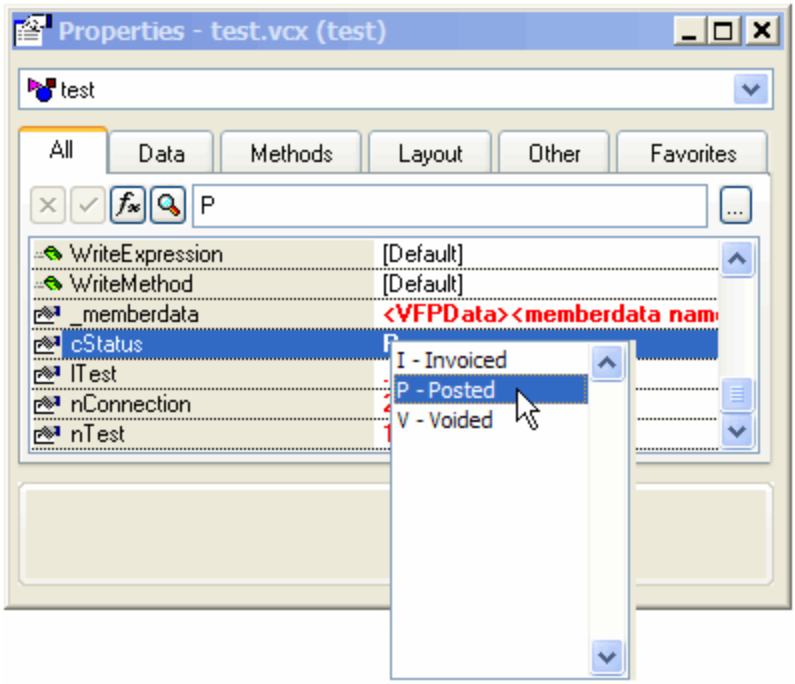

Figure 2. SFPropertyEditorEnumerated displays a listbox of possible values and their descriptions for enumerated properties.

SFPropertyEditorEnumerated needs a list of values to display, so the PropertyEditor method gets the value of the custom editorvalues attribute from the Member Data and parses it into separate value-description sets. PropertyEditor then instantiates SFEnumeratedValueForm, a simple form consisting of a listbox and not much code, and calls its AddValue method to add each value and description to the listbox and its SelectValue method to ensure the current value of the property is selected by default. The trickiest code involves determining where the form should go. Normally, I'd like it to be placed right at the mouse position (MROW() and MCOL() will tell us that) but if the Properties window is dockable, that won't work because the form can't be placed on top of the Properties window. So, the code uses some Windows API functions to figure out where the Properties window is located (we won't look at those methods here), decides whether to put the form to the left or right of the Properties window, and sets the form's Top and Left properties appropriately. Finally, it calls the Show method of the form, and upon return, retrieves the value the user selected from the form and updates the property depending on the data type required.

```
local lcValues, ;
  lcDirectory, ;
  loForm, ;
  laLines[1], ;
  lnLines, ;
  laValues[1], ;
  lnI, ;
  lcValue, ;
  lcDescription, ;
  lnTop, ;
  llDockable, ;
  lnhWnd, ;
  lcBuffer, ;
  lnWLeft, ;
  lnWRight, ;
  lnLeft
with This

* Get the editorvalues attribute.

  .Assert(.oAttributes.GetKey('editorvalues') > 0, ;
    'There is no editorvalues attribute in ' + ;
```

```
    '_MemberData.')
  lcValues = .oAttributes.Item('editorvalues')

* Create the form to display values in and split out
* into the individual values.

  lcDirectory = sys(16)
  lcDirectory = addbs(justpath(substr(lcDirectory, ;
    at(' ', lcDirectory, 2) + 1)))
  loForm      = newobject('SFEnumeratedValueForm', ;
    lcDirectory + 'SFPropertyEditor.vcx')
  lnLines     = alines(laLines, lcValues, 1, ';')
  .Assert(lnLines > 0, 'The editorvalues ' + ;
    'attribute in _MemberData does not contain a ' + ;
    ' valid set of values.')
  dimension laValues[lnLines]
  for lnI = 1 to lnLines
    lcValue       = laLines[lnI]
    lcDescription = strextract(lcValue, ',')
    lcValue       = strextract(lcValue, '', ',')
    laValues[lnI] = lcValue
    loForm.AddValue(lcValue, lcValue + ' - ' + ;
      lcDescription)
  next lnI

* Select the current value.

  loForm.SelectValue(.uCurrentValue)

* Figure out where to put the form. If the Properties
* window is dockable (whether it's currently docked
* or not), we have to put the form beside it since we
* can't go on top of it. In that case, we have to use
* some Windows API functions to locate the Properties
* window and determine its location. If we have
* enough room to put the form to the right of the
* Properties window, do so; otherwise, put it to the
* left.

  lnTop      = min(mrow('', 3), _screen.Height - ;
    loForm.Height)
  llDockable = wdockable('Properties')
  if llDockable
    lnhWnd = .FindWindow(0, 'Properties')
    if lnhWnd <> 0
      lcBuffer = replicate(chr(0), 16)
      if GetWindowRect(lnhWnd, @lcBuffer) <> 0
        lnWLeft  = ctobin(left(lcBuffer, 4), 'rs')
        lnWRight = ctobin(substr(lcBuffer, 9, 4), 'rs')
      endif GetWindowRect(lnhWnd, @lcBuffer) <> 0
      if _screen.Width - loForm.Width > lnWRight
        lnLeft = lnWRight
      else
        lnLeft = lnWLeft - loForm.Width - 5
      endif _screen.Width - loForm.Width > lnWRight
    else
      lnLeft = 0
    endif lnHWnd <> 0
  else
    lnLeft = min(mcol('', 3), _screen.Width - ;
      loForm.Width)
  endif llDockable
  loForm.Top  = lnTop
  loForm.Left = lnLeft

* Display the form and get the selected value.

  loForm.Show()
  lcValue = laValues[loForm.nSelectedValue]
  do case
    case vartype(loForm) <> 'O' or ;
```

```
      loForm.nSelectedValue = 0
    case .cDataType = 'C'
      .UpdateProperty(lcValue)
    otherwise
      .UpdateProperty(int(val(lcValue)))
  endcase
endwith
```

To see this editor in action, open the Test class in Test.VCX and double-click the nConnection or cStatus properties. As we saw earlier, the Member Data for both properties specifies Enumerated as the editor name passed to PropertyEditor.PRG, and Enumerated is registered in PropertyEditors.DBF as the SFPropertyEditorEnumerated class. Both of these properties have an editorvalues attribute that specifies the value and description pairs.

Other property editor thoughts

Another generic property editor, SFPropertyEditorGetFile, is used for properties that contain file names. Create custom attributes called fileext and extdescrip that contain the extension for the file and the description for the extension (for example, "DBF" and "Table") and specify "GetFile" as the editor name passed to PropertyEditor.PRG and the name of the property as the second parameter. We won't look at the code for this editor since it's relatively simple. See the Member Data for the cFileName property in the Test class for an example of how to use it.

Not all property editors you create have to be generic like SFPropertyEditorToggle, SFPropertyEditorEnumerated, and SFPropertyEditorGetFile; your property editors can be specific for a single property. In that case, set the value of cProperty to the name of the property and don't pass a second parameter to PropertyEditor.PRG. Also, a property editor doesn't just have to affect a single property; if the value of one property has an impact on another property, feel free to change both of them at the same time.

Because properties in the Properties window have a limit of 8K characters, and _MemberData can grow quite large quickly when you have a lot of members due to the verbosity of XML, you might want to conserve space in the XML by renaming PropertyEditor.PRG to PE.PRG, putting it in a directory in your VFP path, and referencing a specified editor by ID rather than name. For example, you could use script="PE(1,'PropertyName')" to use SFPropertyEditorToggle, which has an ID of 1, as the editor for the property.

Special thanks to Rick Schummer for testing this, providing valuable suggestions, and, of course, the original idea spark.

Summary

The fact that you can customize and extend the VFP IDE using VFP code makes it possible to improve your productivity immensely. The property editor framework I showed in this article should help you to build editors that reduce the time it takes to create the great applications VFP makes possible.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com