Inside the Object Inspector

Starting with a good framework makes building a new tool much easier, but there are still plenty of challenges.

Tamar E. Granor, Ph.D.

In my last article, I demonstrated the Object and Collection Inspector, a new tool I built to overcome the VFP Debugger's weaknesses in working with collections. This time, I'll open the hood and cover some of the issues I encountered in building the tool.

As I said last time, this tool is a work in progress. Since that article was written, I've added some functionality, so before I dig in, let me show you the additions. The latest version of the tool, as well as the complete source code, is included in the downloads for this article.

New tricks

The most significant change in the current version of the Object Inspector (included in this month's downloads) is the addition of Refresh capability. Originally, the Object Inspector displayed the hierarchy it found and never changed that hierarchy. Now, there's a toolbar with a Refresh button. Click that button (or press F5) and the hierarchy is rebuilt. Figure 1 shows the Object Inspector with the new toolbar.

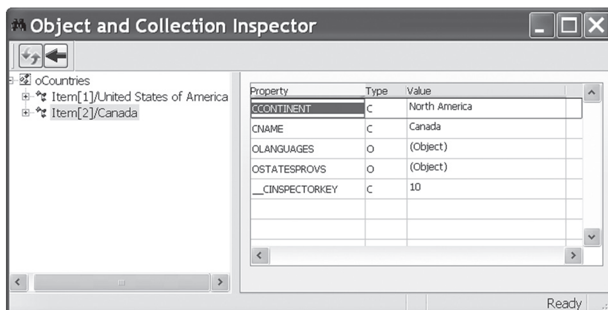

Figure 1. A new toolbar in the Object Inspector gives you access to Refresh and Go Back capabilities.

The Inspector also has two additional ways to navigate. First, you can double-click or press Enter on an object property in the right pane to jump to that object in the treeview. Second, there's now "Go Back" functionality. That is, the tool keeps track of the items to which you navigate and you can return to a previously selected item. The toolbar has a "Go Back" button, but the backspace key offers the same capability.

The Value column of the grid now has Tooltips that show the Value. That's most useful when the Value is larger than the cell.

There's one other functionality change that will be apparent when you run the tool. In the previous version, the treeview opened showing only the root node. Now that node is expanded when the Inspector opens, as in Figure 1.

The initial challenge—failed

When I first envisioned building a collection inspector, I planned to look in memory and find all the collections (or all the objects) and display them in the tool. I even wrote code to parse the results of LIST MEMORY and create a list of objects to display.

Then, I hit reality. Although I could build such a list, I had no way to access those objects. If they were properly declared as local variables, they'd be out of scope inside the tool.

I asked around and no one I spoke to knew of a way around that problem. So I realized that I'd have to pass the root object to the tool, so that it would be in scope.

The starting point

From the beginning, I saw the tool using a Treeview control to represent the hierarchy of objects. I quickly realized that the Explorer form classes that Doug Hennig described in the November, 2008, March, 2009 and May, 2009 issues would be a great starting place. A review of those articles convinced me that the appropriate form class to start with was sfExplorerFormTreeview.

This form class has a treeview in the left panel (it's actually a container object containing a treeview, an imagelist and some other things) and a tabless pageframe in the right panel. The treeview is driven by a cursor; the container has an abstract method called FillTreeViewCursor to fill the cursor. The cursor has a field to indicate which page of the pageframe should be displayed when a particular item is chosen. Doug recommends populating the pages by creating a container class for each.

Filling the cursor

So, the first step in creating my form was to populate the FillTreeViewCursor method of the oTreeViewContainer object. The vast majority of the code I wrote was somehow involved in this process. I needed to fill a cursor with the list of

items to show in the tree. That required traversing the entire object hierarchy and creating a record in the cursor for each collection, collection member, and object property. I wrote recursive code to do the drilldown and quickly found a serious problem.

My code worked for simple hierarchies, but as soon as I tested a hierarchy that included circular references, I had infinite recursion. Obvious, once I thought about it.

The solution is to keep track of what items we've already visited, and before adding an object, check whether we've seen it before. I created a collection to hold those items, and added code to do the checking. I also added an additional field to contain the ID of the referenced item to the cursor filled by the method. Fortunately, Doug's architecture made adding the field as easy as modifying a single property: oTreeViewContainer.cCursorStructure.

Listing 1 shows the code in FillTreeViewCursor. (Code to do some logging for debugging purposes is omitted from all of the following.) The method first creates the collection to hold the list of nodes visited. Then, it examines the root object (passed as a parameter to the form and stored in the form's oRoot property in the Init method). That object is added to the cursor. If the root object is a collection, the form's AddCollectionMembersToTreeViewCursor is called. If it's anything else, we instead call the FindObjectPropertiesForTreeViewCursor method.

Listing 1. The Treeview container's FillTreeViewCursor method starts the process of building the cursor of items to show in the tree.

```
* Set up the collection to track items we've
* seen, to avoid infinite recursion
ThisForm.oItemsVisited = ;
  NEWOBJECT("colItemsVisited", ;
            "ItemsVisited.PRG", "", ThisForm)
ThisForm.oItemsVisited.lAddToObject = ;
  ThisForm.lAddKeys

* First, the root item

IF PEMSTATUS(ThisForm.oRoot, "BaseClass", 5) ;
   AND ;
   UPPER(ThisForm.oRoot.BaseClass) = ;
     "COLLECTION"
  INSERT INTO (This.cCursorAlias) ;
    (ID, TYPE, TEXT, ;
     IMAGE, SORTED, PAGE, ;
     NODEKEY, REFID) ;
    VALUES ;
     ( "ROOT" , "TOP", ThisForm.cRootName, ;
      "Collection", .F., 1, ;
      This.GetNodeKey("TOP", "ROOT"), "")
  ThisForm.oItemsVisited.AddItemVisited(;
    ThisForm.oRoot, "", "ROOT", "TOP")
  ThisForm.;
    AddCollectionMembersToTreeViewCursor( ;
    ThisForm.oRoot, "ROOT", "TOP", ;
    This.cCursorAlias)
ELSE
  INSERT INTO (This.cCursorAlias) ;
    (ID, TYPE, TEXT, ;
     IMAGE, SORTED, PAGE, ;
     NODEKEY, REFID) ;
    VALUES ;
     ( "ROOT" , "TOP", ThisForm.cRootName, ;
      "Class", .F., 2, ;
      This.GetNodeKey("TOP", "ROOT"), "")
  ThisForm.oItemsVisited.AddItemVisited( ;
    ThisForm.oRoot, "", "ROOT", "TOP")

  ThisForm.;
    FindObjectPropertiesForTreeViewCursor( ;
    ThisForm.oRoot, "ROOT", "TOP", ;
    This.cCursorAlias, 1)
ENDIF

RETURN
```

AddCollectionMembersToTreeViewCursor, as its name suggests, loops through a collection and adds each of its members to the cursor. The code is shown in Listing 2.

For each member, the first thing the method does is build a label and a unique key to identify the item. It then figures out whether this is an object and, if so, whether it's a collection. For any object, we check to see whether we've already visited this object. If so, we categorize it as "Previous" and stop drilling down. If not, we add this item to the cursor, and if it is an object, drill down.

Listing 2. The AddCollectionMembersToTreeViewCursor loops through a collection and adds each member to the cursor, drilling down as appropriate.

```
LPARAMETERS oCollection, cParentKey, ;
            cParentType, cAlias, nLevel

IF PCOUNT() < 5
  nLevel = 1
ENDIF

* Make the compiler happy
EXTERNAL ARRAY oCollection

* Now, the members
LOCAL nItem, oObject, cKey, cLabel, ;
      lIsClass, cID, cItemType, nPage, ;
      cRefID, lIsCollection

FOR nItem = 1 TO oCollection.Count
  cLabel = "Item[" + TRANSFORM(m.nItem) + "]"
  cKey = oCollection.GetKey(m.nItem)
  IF NOT EMPTY(m.cKey)
    cLabel = m.cLabel +  "/" + m.cKey
  ENDIF
  cID = m.cParentKey + "/#" + ;
        TRANSFORM(m.nItem)

  lIsClass = ;
    (VARTYPE(oCollection[m.nItem]) = "O")

  * If it's a class, check whether it's a
  * collection
  IF m.lIsClass
    lIsCollection = ;
      (PEMSTATUS(oCollection[m.nItem], ;
                 "BaseClass", 5) AND ;
      UPPER(oCollection[m.nItem].BaseClass);
        = "COLLECTION")
  ENDIF

  * If this is an object, check whether it's
  * already in the items we've visited. If so,
  * categorize it differently and don't
  * drill down.
  cRefID = ""
```

```
  IF m.lIsClass
    cRefID = This.oItemsVisited.GetItemKey(;
                oCollection[m.nItem])
    IF NOT EMPTY(m.cRefID)
      cItemType = "Previous"
      nPage = 4
    ELSE
      IF m.lIsCollection
        cItemType = "Collection"
        nPage = 1
      ELSE
        cItemType = "Class"
        nPage = 2
      ENDIF
    ENDIF
  ELSE
    cItemType = "Nothing"
    nPage = 3
  ENDIF

  INSERT INTO (m.cAlias) ;
    (ID, PARENTID, PARENTTYPE, ;
     TYPE, TEXT, IMAGE, ;
     SORTED, PAGE, REFID, ;
     NODEKEY) ;
    VALUES ;
     (m.cID, m.cParentKey, m.cParentType, ;
      "ITEM", m.cLabel, m.cItemType, ;
      .F., m.nPage, m.cRefID, ;
      This.oTreeViewContainer.GetNodeKey( ;
        "ITEM", m.cID))

  IF m.lIsClass AND m.cItemType <> "Previous"
    This.oItemsVisited.AddItemVisited( ;
      oCollection[m.nItem], m.cParentKey, ;
      m.cID, "ITEM")
  ENDIF

  * Is this member a collection?
  IF m.cItemType = "Collection"
    This.;
      AddCollectionMembersToTreeViewCursor( ;
        oCollection[m.nItem], m.cID, "ITEM", ;
        m.cAlias, m.nLevel + 1)
  ENDIF

  * More likely is that one or more
  * properties are collections or objects.
  IF m.cItemType = "Class"
    This.;
      FindObjectPropertiesForTreeViewCursor( ;
        oCollection[m.nItem], m.cID, "ITEM", ;
        m.cAlias, m.nLevel + 1)
  ENDIF

ENDFOR
```

FindObjectPropertiesForTreeViewCursor (shown in Listing 3) looks inside a given object and adds any properties that reference objects to the cursor. This method uses AMEMBERS() to get a list of the object's properties, and then loops through that list. If the property is an object, it builds a reference to that object. Then, it calls the appropriate method to add that object to the cursor and drill down.

This method skips over two kinds of objects it encounters, COM objects and the GDIPlusX root object. That is, the tool can't drill into those object types.

Listing 3. FindObjectPropertiesForTreeViewCursor goes through the list of properties for an object. If any refer to objects, it calls the appropriate method to add them to the cursor.

```
LPARAMETERS oObject, cParentKey, ;
            cParentType, cAlias, nLevel

LOCAL aProps[1], nPropCount, nProp, ;
      oChildObject

nPropCount = AMEMBERS(aProps, m.oObject, 0)
FOR nProp = 1 TO m.nPropCount
  IF TYPE("oObject." + aProps[m.nProp]) = "O"
    oChildObject = ;
      EVALUATE("oObject." + aProps[m.nProp])
    * Want to omit COM and null objects
    IF NOT ISNULL(m.oChildObject)
      DO CASE
      CASE PEMSTATUS(m.oChildObject, ;
                     "BaseClass", 5) ;
           AND ;
           UPPER(oChildObject.BaseClass) =;
           "COLLECTION"
       This.;
       AddCollectionPropertyToTreeViewCursor(;
         m.oChildObject, ;
         aProps[m.nProp], m.cParentKey, ;
         m.cParentType, m.cAlias, ;
         m.nLevel + 1)
      CASE ALLTRIM(COMCLASSINFO( ;
             m.oChildObject, 5)) = "1"
        IF NOT PEMSTATUS(m.oChildObject, ;
                         "Name", 5) ;
           OR ;
           NOT UPPER(m.oChildObject.Name) = ;
               "XFCSYSTEM"
          * Omit GDIPlusX stuff to avoid
          * trouble
           This.;
           AddObjectPropertyToTreeViewCursor(;
             m.oChildObject, ;
             aProps[m.nProp], m.cParentKey, ;
              m.cParentType, m.cAlias, ;
              m.nLevel + 1)
        ENDIF
      OTHERWISE
        * COM object. Don't add it.
      ENDCASE
    ENDIF
  ENDIF
ENDFOR
```

Two more methods actually put the information about the objects and collections identified into the cursor. They're AddObjectPropertyToTreeViewCursor, shown in Listing 4, and AddCollectionPropertyToTreeViewCursor, similar enough that the code is omitted here. Each of them checks whether we've already added this item. If not, it adds a record and then drills down, calling the appropriate method.

Listing 4. AddObjectPropertyToTreeViewCursor adds a property representing an object to the cursor and drills down.

```
LPARAMETERS oObject, cPropName, cParentKey, ;
            cParentType, cAlias, nLevel

LOCAL cLabel, cID, cItemType, nPage,
LOCAL cName, cRefID

IF PEMSTATUS(oObject, "Name", 5)
  cName = oObject.Name
ELSE
  cName = "UNNAMED"
ENDIF
```

```
cLabel = m.cPropName
cID = m.cParentKey + "/" + m.cPropName

* Check whether we've seen it before, and if
* so, categorize appropriately and avoid
* drilldown

cRefID = This.oItemsVisited.GetItemKey( ;
           m.oObject)
IF NOT EMPTY(m.cRefID)
  cItemType = "Previous"
  nPage = 4
ELSE
  cItemType = "Class"
  nPage = 2
ENDIF

INSERT INTO (m.cAlias) ;
  (ID, PARENTID, PARENTTYPE, ;
   TYPE, TEXT, IMAGE, ;
   SORTED, PAGE, REFID, ;
   NODEKEY) ;
  VALUES ;
  (m.cID, m.cParentKey, m.cParentType, ;
   "PROPERTY", m.cLabel, m.cItemType, ;
   .F., m.nPage, m.cRefID, ;
   This.oTreeViewContainer.GetNodeKey( ;
     "PROPERTY", m.cID))

IF m.cItemType = "Class"
  This.oItemsVisited.AddItemVisited( ;
    m.oObject, m.cParentKey, m.cID, ;
    "PROPERTY")

  * Now add its members
  This.FindObjectPropertiesForTreeViewCursor(;
    m.oObject, m.cID, "PROPERTY", m.cAlias, ;
    m.nLevel + 1)
ENDIF

RETURN
```

Tracking what we've already seen

As I described above, in order to avoid infinite recursion, we need a way to know that we've already added an item to the cursor. What makes this difficult is that we don't know anything about the items themselves. Since this is a developer tool, they can be anything at all. So we can't just grab some natural identifier of the items and keep track of that. Instead, we have to keep track of the items themselves.

To do that, I created a class called cusItemVisited with the following custom properties:

- oObject: Holds a reference to the object we've visited;
- cParentKey: The key in the treeview cursor for the parent of the object the first time we encountered it;
- cID: The ID for the object in the cursor;
- cType: The type of object.

A collection class named colItemsVisited holds the cusItemVisited objects. As you'd expect, the AddItemVisited method creates and adds a cusItemVisited object; the code is shown later in this section (Listing 7), after discussing an alternative approach.

The core method of the collection is GetItemKey (shown in Listing 5); you pass an object and it searches the collection to see whether the object is there. If it is, it returns the key (composed of the type and ID, separated by the character "~") for that object.

Listing 5. The GetItemKey method of the colItemsVisited collection lets you know whether a given object has already been added to the cursor.

```
LOCAL oItem, lFound, cID, cType, cKey

lFound = .F.

FOR EACH oItem IN This FOXOBJECT
  IF oItem.oObject = m.oObject
    lFound = .T.
    cID = oItem.cID
    cType = oItem.cType
    EXIT
  ENDIF
ENDFOR

IF m.lFound
  cKey = TRIM(m.cType) + "~" + TRIM(m.cID)
ELSE
  cKey = ""
ENDIF

RETURN m.cKey
```

The code that calls this method (see, for example, Listing 4) uses the returned key to figure out what to do. If it's empty, the item isn't already in the cursor, so the code adds it as the appropriate type and then drills down. If the key is not empty, we know that the object is already in the cursor, so the item is added to the cursor with a type of "Previous," which turns it into a backlink.

This strategy works and the tree is properly populated. However, because it uses a brute force search, for large hierarchies, it becomes exceptionally slow. The application that led me to build this tool can have thousands of items in the hierarchy I want to display. Populating the cursor was taking minutes.

So I came up with an alternate strategy. However, because it involves modifying the objects in the original object hierarchy, I made it optional. The new approach adds a property containing the key to every object it visits. Then, GetItemKey can simply check for the presence of that property and return its value, if it's there. The GetByKey method looks up an item in the collection using its key. Listing 6 shows the code in a separate branch of GetItemKey that handles this. This code goes after the initialization of lFound. The FOR loop goes into the ELSE branch.

Listing 6. To speed up population of the treeview, you can allow the tool to add a property to each object in the hierarchy. This code in GetItemKey uses that property to quickly return the key.

```
IF This.lAddToObject
  IF PEMSTATUS(m.oObject, ;
               "__cInspectorKey", 5)
    oItem = This.GetByKey( ;
              oObject.__cInspectorKey)
```

```
    IF NOT ISNULL(m.oItem)
      cID = oItem.cID
      cType = oItem.cType
      lFound = .T.
    ENDIF
  ENDIF
ELSE
  * Continue with the brute force search shown
  * in Listing 5
ENDIF
```

To make this strategy work, of course, we need code that adds the property to each object the first time it's encountered. That code is in the AddItemVisited method. The complete code for that method, showing both approaches, is in Listing 7.

Listing 7. The AddItemVisited method of colItemsVisited accepts information about an object in the hierarchy and adds an item to the collection for that object. If the lAddToObject flag is set, it also adds a property to the object to make it easier to see whether we've already visited the object.

```
LPARAMETERS oObject, cParentKey, cID, cType

LOCAL oItem, cName
LOCAL cObjectKey

IF This.lAddToObject
  cObjectKey = This.GetNewKey()
  IF PEMSTATUS(m.oObject, ;
               "__cInspectorKey", 5)
    oObject.__cInspectorKey = m.cObjectKey
  ELSE
    ADDPROPERTY(m.oObject, ;
                "__cInspectorKey", ;
                m.cObjectKey)
  ENDIF
ENDIF

oItem = CREATEOBJECT("cusItemVisited", ;
          m.oObject, m.cParentKey, ;
          m.cID, m.cType)

IF This.lAddToObject
  This.Add(m.oItem, m.cObjectKey)
ELSE
  This.Add(m.oItem)
ENDIF

RETURN
```

By default, the Object Inspector uses the new strategy, since it provides better performance. To use the original strategy that leaves your objects untouched, pass .T. for the optional third parameter when calling the tool.

Filling the right-hand pane

As noted earlier, the sfExplorerFormTreeView class contains code to display a particular page in the right panel's pageframe in response to the user's selection from the treeview. To take advantage of this ability, I set PageCount to 4 for the pageframe in the right pane.

Then, the main thing I needed to do was create a container class to put on each of the pages. The four different page types I identified were:

- 1: Collection
- 2: Object
- 3: Scalar item
- 4: Back reference to an object elsewhere in the hierarchy

As I started to create the container classes, it quickly became apparent that there was code I'd need on every page, so I created a generic container class with the necessary code and subclassed the individual containers from that one.

The key thing needed on every page is a way to figure out what to display. I added a property, oCurrentObject, to hold a reference to the relevant object. The Refresh method contains just one line of code, setting this property to the result of a call to the custom ParseID method. ParseID, shown in Listing 8, uses the form's cCurrentNodeID to build a reference to the corresponding object.

Listing 8. When we land on a particular page of the pageframe, we need to figure out what to display. This method uses the form's ID for the current treeview node to figure out what object it refers to.

```
LOCAL aLevels[1], nLevels, nLevel, ;
      cHierarchy, oObject

nLevels = ALINES(m.aLevels, ;
                 ThisForm.cCurrentNodeID, "/")

cHierarchy = "ThisForm.oRoot"

FOR nLevel = 2 TO m.nLevels
  IF LEFT(aLevels[m.nLevel], 1) = "#"
    * It's an item in the current collection
    cHierarchy = m.cHierarchy + ".Item[" + ;
                 SUBSTR(aLevels[m.nLevel], 2);
                 + "]"
  ELSE
    * It's a property of the current object
    cHierarchy = m.cHierarchy + "." + ;
                ALLTRIM(aLevels[m.nLevel])
  ENDIF
ENDFOR

TRY
  oObject = EVALUATE(m.cHierarchy)
CATCH
  oObject = .null.
ENDTRY

RETURN m.oObject
```

The individual container classes contain very little code. All except the back reference class (cntBackReference) have additional code in Refresh to gather the data to be displayed.

Both cntCollectionInfo (used on page 1) and cntObjectInfo (used on page 2) contain a grid to show the properties of the object and their values. In both cases, Refresh calls a method of the grid to fill it.

The grid is based on a class called grdProperties. It has a custom method, CreatePropertyCursor, called by the Init method. CreatePropertyCursor

creates a cursor with three columns and stores its alias in a property, cCursorAlias. The key method of the grid is FillPropertyCursor, shown in Listing 9, which accepts an object as parameter, uses AMEMBERS() to get a list of the properties of the object and fills the cursor with that list.

Listing 9. The FillPropertyCursor method of grdProperties collects a list of properties of an object and puts them into a cursor along with their values.

```
LPARAMETERS oObject

IF RECCOUNT(This.cCursorAlias) > 0
  ZAP IN (This.cCursorAlias)
ENDIF

LOCAL aProps[1], nPropCount, nProp,
LOCAL cValue, cType

This.oObject = m.oObject

IF NOT ISNULL(m.oObject)
  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
ENDIF

GO TOP IN (This.cCursorAlias)

RETURN
```

The grid also contains code to resize its columns proportionally when the grid is resized.

Handling back references

As I described earlier, each object in the hierarchy is fully displayed only once. If the same object is referenced more than once in the hierarchy, all references except the first show a simple message that lets you jump to the original reference. Figure 2 shows an example.

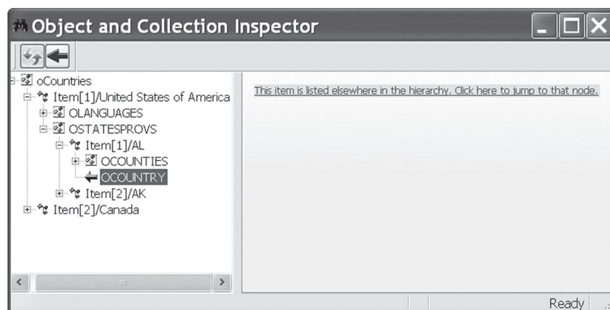

Figure 2. When an object is referenced more than once in the hierarchy, references after the first let you jump to the full display for the item.

cntBackReference contains a single label with FontUnderline set to .T. and forecolor set to blue (0, 0, 255). The Click method of the label contains the code shown in Listing 10, which gets the key for the right item, makes sure it's expanded and then selects it.

Listing 10. The Click method of the label on the Back Reference page finds the place in the treeview where the current item is fully displayed and displays that node.

```
LOCAL cParentKey

cParentKey = ;
  ThisForm.GetKeyForCurrentNodeRefID()
ThisForm.ExpandAllParents(m.cParentKey)
ThisForm.oTreeViewContainer.SelectNode( ;
  m.cParentKey)

RETURN
```

Getting the key for the right item is actually fairly simple, because it's stored in the field I added to the treeview cursor. So GetKeyForCurrentNodeRefID just SEEKs the right record in the cursor and returns its RefID field.

The treeview container method SelectNode selects the item whose key you pass. However, it only works if the item is current displayed. If the specified node is in a portion of the tree that hasn't been expanded, SelectNode doesn't do anything.

So I added a method to the form to expand all the ancestor nodes of a specified node, starting from the first unexpanded node and working down to the parent node of the specified node. ExpandAllParents is shown in Listing 11.

Listing 11. Before selecting an item in the treeview, we need to ensure that it's displayed by expanding each of its ancestors.

```
LPARAMETERS cKey

LOCAL lExpanded, aKeys[1], nToBeExpanded,
LOCAL cAlias
* Work upwards from the key, until we find a
* node that was previously expanded.

lExpanded = .F.
cAlias = This.oTreeViewContainer.cCursorAlias

* Put the first item into the array
aKeys[1] = m.cKey
nToBeExpanded = 1

* Go to record for the specified key
SEEK m.cKey ORDER NODEKEY IN (m.cAlias)

DO WHILE NOT m.lExpanded

  TRY
    oItem = ;
     This.oTreeViewContainer.oTree.Nodes[ ;
       m.cKey]
    * If we're still here, the item exists and
    * thus must have been expanded.
    lExpanded = .T.

  CATCH
    * Find the parent of this item.
    cParentID = ALLTRIM(EVALUATE( ;
```

```
            m.cAlias + ".ParentID"))
    IF SEEK(m.cParentID, m.cAlias, "ID")
      cKey = ALLTRIM(EVALUATE( ;
             m.cAlias + ".NodeKey"))

      * Add this item to the list to expand
      nToBeExpanded = m.nToBeExpanded + 1
      DIMENSION aKeys[m.nToBeExpanded]
      aKeys[m.nToBeExpanded] = m.cKey

    ELSE
      * Get out of here. There's nothing more
      * we can do.
      lExpanded = .T.
    ENDIF

  ENDTRY
ENDDO

* Now go backwards through the array and
* expand
FOR nItem = m.nToBeExpanded TO 1 STEP -1
  This.oTreeViewContainer.TreeExpand( ;
    aKeys[m.nItem], .T.)
ENDFOR

RETURN
```

Making the grid more useful

The properties grid shown on the Collection and Object pages has several useful features. First, you can double-click on the value of a property to open a form showing the value in a larger editbox. The editbox also provides its own value as a tooltip. Finally, double-clicking or pressing Enter on any property that's an object jumps to that object in the treeview.

Zooming a property value takes advantage of functionality Doug provided in his classes. I subclassed Doug's sfEditBox class to create edtDynamicTooltip. Doug's class has built-in zooming. All I had to do was set the cZoomClass and cZoomLibrary properties to point to Doug's sfEditBoxZoomForm class. I set the cZoomFormCaption property to "Zoom Property Value." Because I don't want users to try to edit the Values, I also put code in the editbox's SetZoomFormProperties class to make the zoomed editbox readonly.

Providing tooltips for the Value column, my main reason for subclassing, was much harder. In all versions of VFP except VFP9 SP1, only grid-level tooltips display, rather than tooltips for the items within the grid. To provide a tooltip based on the editbox value requires code in the grid and the editbox.

I added an Access method to the grid's ToolTipText property; the code is shown in Listing 12. If the control in the column has a tooltip, it returns that; if not and the column has one, it returns the column's ToolTip.

Listing 12. This code in the grid's ToolTipText_Assign method checks the column and the control in the column for a tooltip.

```
LOCAL cToolTip, aMousePos[1], oColumn, ;
      oControl
cToolTip = ""

IF AMOUSEOBJ(aMousePos) > 0
  oColumn = aMousePos[1]
  IF NOT ISNULL(m.oColumn) AND ;
     UPPER(oColumn.BaseClass) = "COLUMN"
    * First, grab column-level tip in case we
    * don't find something below
    cToolTip = oColumn.ToolTipText

    * Now, look for the right control. It
    * should be the last control.
    IF oColumn.Objects.Count > 0
      oControl = ;
        oColumn.Objects[oColumn.Objects.Count]
      IF NOT EMPTY(oControl.ToolTipText) OR ;
         PEMSTATUS(m.oControl, ;
                   "ToolTipText_Access", 5)
        cToolTip = oControl.ToolTipText
      ENDIF
    ENDIF
  ENDIF
ENDIF

RETURN m.cToolTip
```

To return the contents of the Value column for the row where the mouse is located requires an Access method for the editbox's ToolTipText, as well. Shown in Listing 13, it uses GridHitTest to figure out which row we're over, sets focus to that cell, and returns its value.

Listing 13. The editbox's ToolTipText_Access method figures out which row we're in, and returns the right value.

```
LOCAL nRow, nCol
LOCAL nGridComp, nRelRow, nRelCol
LOCAL cTip

nRow = MROW(_screen.ActiveForm.Name, 3)
nCol = MCOL(_screen.ActiveForm.Name, 3)
This.Parent.Parent.GridHitTest(m.nCol, ;
  m.nRow, @nGridComp, @nRelRow, @nRelCol)

* Activate the relevant cell
This.Parent.Parent.ActivateCell( ;
  m.nRelRow, m.nRelCol)
cTip = This.Value

RETURN m.cTip
```

To provide the ability to jump to an item from the property in the grid, I added a method called JumpToItem to the form. It's called from the DoubleClick and KeyPress methods of the textbox in the grid's Property column. JumpToItem, shown in Listing 14, uses methods of the TreeViewContainer object to find the right node and select it.

Listing 14. The form's JumpToItem method accepts the name of a property of the currently displayed node and finds and selects the corresponding child node.

```
LPARAMETERS cName

LOCAL oCurrentNode, oChildren, oChild, lFound

* Get info about current node
oCurrentNode = ;
This.otreeViewContainer.GetTypeAndIDFromNode(;
  This.otreeViewContainer.oSelectedNode)
```

```
* Get children of current node
oChildren = CREATEOBJECT("Collection")
This.oTreeViewContainer.GetChildNodes( ;
  oCurrentNode.Type, oCurrentNode.ID, ;
  oChildren)

* Find the child we're interested in
lFound = .F.
FOR EACH oChild IN m.oChildren FOXOBJECT
  IF UPPER(oChild.Text) = UPPER(m.cName)
    lFound = .T.
    EXIT
  ENDIF
ENDFOR

IF m.lFound
  * Make sure current node is expanded
  This.oTreeViewContainer.TreeExpand( ;
    This.otreeViewContainer.oSelectedNode, ;
    .T.)

  This.oTreeViewContainer.SelectNode(m.oChild)
ENDIF

RETURN
```

Since the grid always shows the properties of the currently selected item, we know that the node we want to select must be a child of that item. The name of the property that points to the object is passed as a parameter.

The method first gets identifying information for the currently selected node. We then retrieve a list of that node's child nodes. We loop through that list to find the right node. If we find it, we make sure the current node is expanded in the tree, and then select the appropriate child.

Adding a toolbar

Adding a toolbar to the Object Inspector was easy. The sfExplorerForm class includes cToolBarClass and cToolbarLibrary parameters. Setting those to point to my toolbar class was all I had to do to get the toolbar displayed at the top.

To create the toolbar, I subclassed Doug's sfToolbar class, and added the buttons I wanted, based on the sfCommandButton class. In each case, the code in Click is just a call to a custom method of the form.

For the Go Back button, I also added code to the Refresh method to disable it when the stack of previous choices is empty.

Refreshing the Inspector

Once I had the toolbar available, refreshing the treeview to reflect an updated object hierarchy turned out to be easy as well. Refreshing is just rebuilding the tree.

I added a custom RefreshTree method to the form. It calls the TreeViewContainer's LoadTree method and then refreshes the form.

Implementing "Go Back"

sfExplorerFormTreeView includes "Go Back" functionality for the treeview, that is, the ability to return to the previously selected node. So, implementing it for the Object Inspector was just a matter of connecting to the built-in code.

I added a method called GoBack to the form. It calls the TreeViewContainer's GoBack method.

Lessons learned

Perhaps the key lesson from building the Object Inspector is how much higher you can get when standing on someone else's shoulders. Because I was starting with a boatload of functionality, I was able to focus on the specific needs of my project without having to build all the necessary infrastructure. Being able to add a toolbar or provide a zooming editbox with no code was wonderful.

However, that doesn't mean it was always easy. Working with someone else's code can be challenging, even when it's documented as well as Doug's (where there were not only comments in the code, including a detailed About method for each class, but the FoxRockX articles as well). I spent a fair amount of time looking through the list of methods, reading code and reviewing documentation to figure out how to do some of the things I needed.

In addition, as I often do, I used the Debugger extensively to experiment along the way. Among other things, those experiments led me to follow the model of other VFP tools and create a public variable (_oInspector) to point to the running Inspector.

As I said in my last article, the Object Inspector is a work in progress. In fact, I added functionality in the course of writing this article. I hope my next lessons will be about working with a community team to improve a tool.

Author Profile

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. She currently focuses on working with other developers through consulting and subcontracting. 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 VisualFoxPro and Taming Visual FoxPro's SQL. Her latest collaboration is Making Sense of Sedna and SP2. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional and 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.