VFP: Ideal for tools

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

We've had developer tools written in the FoxPro language since the days of FoxPro 2.0. Quite a few come with VFP, and the community has built lots more. In this session, we'll explore the Visual FoxPro language features that make building developer tools possible. We'll look at how the VFP language allows you to dig into projects, classes, data and code. Examples will be drawn from a variety of VFP tools, including those in VFPX.

When FoxPro 2.0 shipped in 1991, it included two new tools written in FoxPro’s programming language: GenScrn and GenMenu, which translated screen designs and menu designs, respectively, into FoxPro code. While these may not have been the first developer tools written in FoxPro, they mark a turning point because they were an essential part of developing FoxPro applications and because the programming language was extended to make it possible to write them. Since then, every version of FoxPro and Visual FoxPro has included multiple tools written in FoxPro, and has included commands, functions or other capabilities added specifically to make tool-writing easier. Since then, also, the FoxPro community has created hundreds or thousands of tools written in FoxPro to manipulate FoxPro.

Starting in VFP 6, the source code for all the tools written in the FoxPro language that come with VFP has been included with the product (in Tools\XSource\XSource.ZIP). VFP 9 includes more than 20 so-called "Xbase tools," from the Class Browser to the three reporting applications that help extend the Report Designer. All this source code both enables the VFP community to modify and extend the existing tools, and to see how particular things were done. (In fact, the Xbase tools are now part of VFPX, so that extensions can be managed and distributed.)

The set of language capabilities that enable tool creation is extensive. In this session, we'll look at those capabilities, grouped by what kind of thing they manipulate. To demonstrate them, we'll look at code from a number of different VFP tools. Most of the code in this session was written by people other than me.

The Array functions

There are a large number of functions in VFP that collect information and put the results into an array. Before actually exploring any specific tool-building language features, let's take a quick look at what all these array functions have in common.

First, all of them have names beginning with the letter "A," such as AFIELDS() or AVCXClasses().

Second, each expects an array as the first parameter. The information collected by the function is put into the array, which is created, if necessary, and redimensioned as needed.

Finally, these functions almost all return the number of rows in the resulting array.

For more information on the array functions or on array-handling in general in VFP, see the white paper "You Need Arrays," available on the Session Materials page (http://tomorrowssolutionsllc.com/session_materials.htm) of my website.

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

File type	Table (DBF) extension	Memo (FPT) extension
Menu	MNX	MNT
Project	PJX	PJT
Report	FRX	FRT

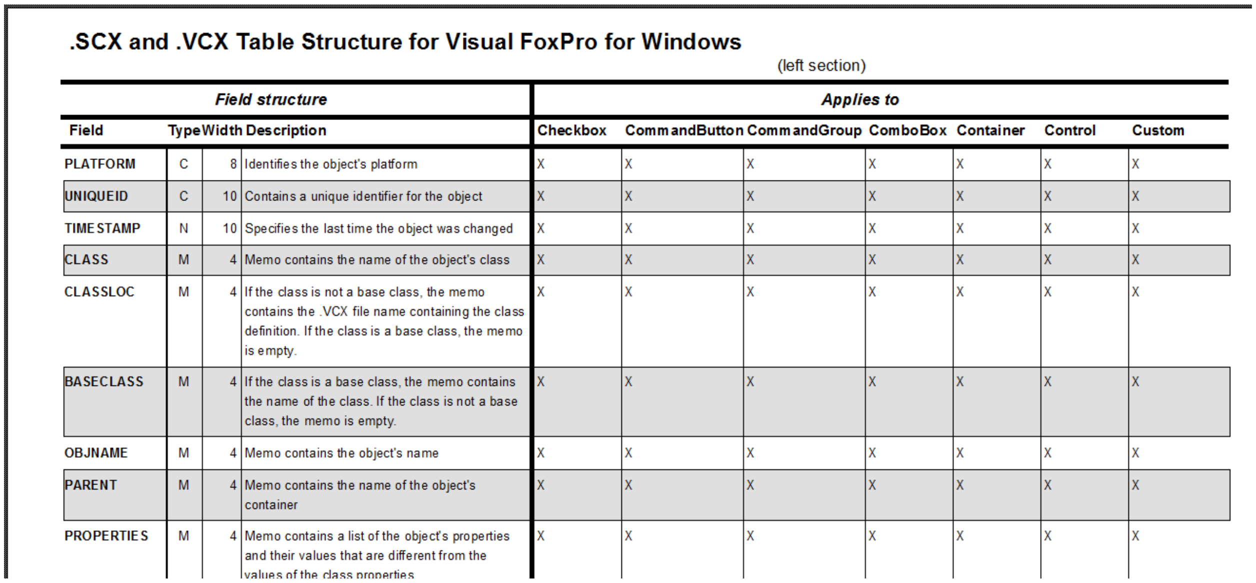
Having all these definitions in tables means that you can use the normal data-handling tools of VFP to work on them. For example, the method in Listing 1 comes from the VFPX PEM Editor tool (specifically, the pemeditor_idex class); it checks whether a specified class is in a specified class library.

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.



.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.

Digging into Data

The big thing that sets VFP apart from conventional programming languages is its native database engine. Not surprisingly, there are a whole slew of functions that let you explore databases and tables and find out which are currently in use.

Even if you're working with a SQL database, VFP includes functions to help you understand the available data structures.

Determining VFP table structures

FoxPro has had the ability to determine the fields in a table from early days. The AFields() function first appeared in FoxPro 2.0; it puts the structure of a table into an array with one row for each field. The content of the array has expanded over the years, as tables gained new capabilities. In VFP 9, the resulting array has 18 columns, though only the first six are relevant for free tables. The function returns the number of fields in the table.

The syntax for AFields() is shown in Listing 2. Listing 3 shows most of the code from the Thor tool, Create SQL from cursor. It uses AFields() to get a list of the fields in the specified cursor and then loops through the resulting array, building a CREATE CURSOR statement. The result of this code, applied to the Northwind Products table, is shown in Listing 4.

Listing 2. AFields() fills an array with information about the fields in a table. It can work on the current workarea or a specified alias or workarea.

```
nFields = AFIELDS( ArrayName [, cAlias | nWorkarea] )
```

Listing 3. This code, drawn from the Thor tool Create SQL from cursor, shows a fairly typical use of AFields().

```
lnFields = afields(laFields)
```

```
lcSQL    = ''
for lnI = 1 to lnFields
     lcType = laFields[lnI, 2]
     lcSQL  = lcSQL + ;
            iif(empty(lcSQL), 'create cursor TEMP ;' + ccCR + '(', ', ;' + ccCR) + ;
            laFields[lnI, 1] + ' ' + lcType
     do case
            case lcType $ 'CVQ'
                 lcSQL = lcSQL + '(' + transform(laFields[lnI, 3]) + ')'
            case lcType $ 'NF'
                 lcSQL = lcSQL + '(' + transform(laFields[lnI, 3]) + ',' + ;
                      transform(laFields[lnI, 4]) + ')'
            case lcType = 'B'
                 lcSQL = lcSQL + '(' + transform(laFields[lnI, 4]) + ')'
     endcase
next lnI
```

Listing 4. The result produced by the Thor tool Create SQL from cursor, applied to the Northwind Products table.

```
create cursor TEMP ;
  (PRODUCTID I, ;
  PRODUCTNAME C(40), ;
  SUPPLIERID I, ;
  CATEGORYID I, ;
  QUANTITYPERUNIT C(20), ;
  UNITPRICE Y, ;
  UNITSINSTOCK I, ;
  UNITSONORDER I, ;
  REORDERLEVEL I, ;
  DISCONTINUED L)
```

The other key thing we may want to know about a table is what indexes it has. Although this information has been available since early FoxPro days, collecting it got a lot easier in VFP 7 with the addition of the ATagInfo() function. As its name implies, this function puts information about index tags into an array. The resulting array has one row for each tag and six columns indicating the name, type, key, filter, order and collate sequence for the tag. The function returns the number of tags; its syntax is shown in Listing 5.

Listing 5. ATagInfo() fills an array with information about index tags.

```
nTags = ATAGINFO( ArrayName [, cIndexName [, cAlias | nWorkarea] )
```

Note that the second parameter to ATagInfo() is the name of an index file. This allows you to collect information on a subset of a table's indexes, in the odd case where they're not all in the table's structural index (CDX). While you're unlikely to use that parameter, you need to be aware of it when you want to specify the alias or workarea of the table. To skip over the cIndexName parameter, pass the empty string, as in Listing 6.

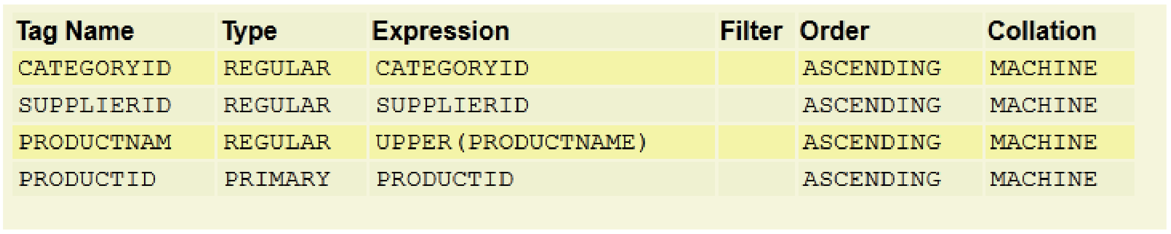
Listing 6. To specify the alias or workarea for ATagInfo(), you must pass something for the cIndexName parameter. Pass the empty string to include all tags in the result.

```
nTags = ATAGINFO(aTags, '', 'MyAlias')
```

Listing 7 shows a block of code from the Thor Schema tool. The section shown creates the list of indexes for the schema. That section of the output, for the Northwind Products table, is shown in Figure 2.

Listing 7. This block of code, from Thor's Schema tool, uses ATagInfo() to collect the list of tags for the table. It then loops through and produces output describing each.

```
If Not This.lView
  Dimension aTags[1]
  iTags = Ataginfo(aTags)
  If iTags = 0
    \ No Structural Index Tags
  Else
    \ <h4>Indexes:</h4><table>

    \ <table id='tblIndices'><tr><th>Tag Name</th> <th>Type</th> <th>Expression</th> 
<th>Filter</th> <th>Order</th> <th>Collation</th></tr>
    For X = 1 To iTags
      \<tr>
      For Y = 1 To 6
        \ <td> <<aTags[X,Y]>>  </td>
      Next
      \</tr>
    Next
  Endif
  \</table>
Endif
```



Tag Name	Type	Expression	Filter	Order	Collation
CATEGORYID	REGULAR	CATEGORYID		ASCENDING	MACHINE
SUPPLIERID	REGULAR	SUPPLIERID		ASCENDING	MACHINE
PRODUCTNAM	REGULAR	UPPER(PRODUCTNAME)		ASCENDING	MACHINE
PRODUCTID	PRIMARY	PRODUCTID		ASCENDING	MACHINE

Figure 2. The tag section of the output from Thor's Schema tool, applied to the Northwind Products table.

Exploring VFP database structures

VFP also includes commands that let you discover the structure of a database, including what tables, views and connections it contains, and the details of those items. The ADBObjects() function is a one-stop shop for finding out what's in a database, while the DBGetProp() function lets you look up the details of database contents.

ADBObjects() fills an array with a list of one kind of thing in a database. You pass a parameter indicating whether you're interested in tables, views, connections or relations. Listing 8 shows the syntax; cInfoType is one of this list: "TABLE", "VIEW", "CONNECTION", "RELATION". For everything other than relations, the array created has a single column with the names of the specified objects. For relations, the function creates a five-column array providing the names of the tables involved, the tags used to create the relation, and a string indicating whether there are any relational integrity constraints based on this relation.

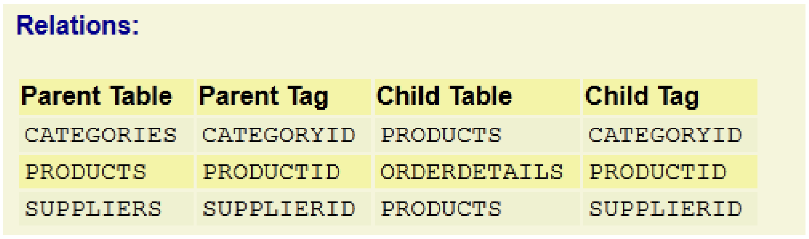
Listing 8. Use ADBObjects() to explore database contents.

```
nItemCount = ADBObjects( ArrayName, cInfoType)
```

ADBObjects() is also used in the Thor Schema, to collect information about the relations in the database. Listing 9 shows that part of the code, while Figure 3 shows that section in the output for the Northwind Products table.

Listing 9. This block of code from Thor's Schema tool reports on relationships in the database.

```
iDbObjects = Adbobjects(aDb,"RELATION")
If Ascan(aDb,Upper(This.cAlias))>0
\ <h4>Relations:</h4><table>
\ <tr><th>Parent Table</th><th>Parent Tag</th><th>Child Table</th><th>Child
Tag</th></tr>
  For X = 1 To iDbObjects
    If aDb[X,1]=Upper(This.cAlias) Or aDb[X,2]=Upper(This.cAlias)
\<tr> <td> <<aDb[X,2]>> </td><td><<aDb[X,4]>> </td><td><<aDb[X,1]>>
</td><td><<aDb[X,3]>> </td></tr>
    Endif
  Next
Endif
```


Relations:

Parent Table	Parent Tag	Child Table	Child Tag
CATEGORIES	CATEGORYID	PRODUCTS	CATEGORYID
PRODUCTS	PRODUCTID	ORDERDETAILS	PRODUCTID
SUPPLIERS	SUPPLIERID	PRODUCTS	SUPPLIERID

Figure 3. Thor's Schema tool uses ADBObjects() to collect information about relations involving the specified table.

DBGetProp() lets you find the value of a specific characteristic of an object in a database. You pass the name of the object, the type of object, and the name of the property you're interested in, and the function returns the value of that property. You can ask about anything from the default value of a field to the primary key of a table to the SQL that defines a view. Listing 10 shows the syntax for the function.

Listing 10. DBGetProp() lets you explore the properties of a database and its contents.

```
uPropertyValue = DBGetProp( cItem, cItemType, cProperty )
```

The possible values for cItemType are: “CONNECTION”, “DATABASE”, “FIELD”, “TABLE”, and “VIEW”. The list of values you can pass for cProperty varies with the item type. There’s a complete list of properties in the Help topic “DBGETPROP( ) Function.”

Listing 11 shows a small block of code found in the DoSearch method of the RefSearchDatabase class of Code References. It uses ADBObjects() and DBGetProp() to add the tables in a database to the list of places to search.

Listing 11. This method inside Code References adds the list of tables in a database to the list of places to be searched.

```
* Add tables in DBC to search list
m.nCnt = ADBOBJECTS(aDBList, "TABLE")
FOR m.i = 1 TO m.nCnt
  TRY
    m.cTableName = DBGETPROP(aDBList[m.i], "TABLE", "PATH")
    THIS.AddFileToSearch(FULLPATH(m.cTableName, ADDBS(THIS.Folder)))
  CATCH
    * ignore error (we might possibly get one on DBGETPROP)
  ENDTRY
ENDFOR
```

DBGetProp() has a sibling, DBSetProp(), that lets you set properties of the items in a database. It’s handy for tools that manage databases. The syntax, shown in Listing 12, is similar to DBGetProp()’s, but there’s a fourth parameter to provide the new value of the specified property.

Listing 12. DBSetProp() lets you change the properties of an object in a database.

```
lSuccess = DBSETPROP( cItem, cItemType, cProperty, uNewValue )
```

Be aware that some properties of database objects can’t be changed by DBSetProp(); the list of properties in Help indicates, for each, whether it’s read-only or read-write. Those that can’t be changed be DBSetProp() generally have another command to set them. For example, the SQL property of a view is set by the CREATE SQL VIEW command.

Listing 13 is drawn from the DoReplace method of the RefSearchDatabase class of Code References. It uses DBSetProp() to change the Comment for a database.

Listing 13. Use DBSetProp() to set those properties of database objects that aren’t handled by other, more specific, commands.

```
DBSETPROP(JUSTSTEM(THIS.Filename), "DATABASE", "Comment", cNewText)
```

Exploring SQL database structures

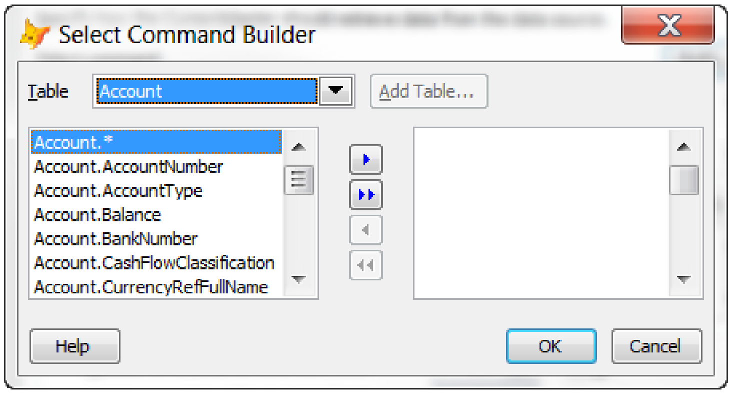
VFP also includes some functions that let you explore the structure of a SQL database. What’s particularly nice about these functions is that they work for pretty much any SQL database, and the results are structured the same way no matter which back-end you’re talking to. Both of these functions require that you’ve already connected to the database, as they expect the handle as the first parameter.

SQLTables() returns a list of tables in the database. You can optionally limit the result to just tables, just views or just system tables. The syntax for SQLTables() is shown in Listing 14. The function returns 1 if it's done, 0 if it's still executing (only possible if you're executing it asynchronously), or a negative value if an error occurred.

Listing 14. SQLTables fills a cursor with the list of tables in a SQL database.

```
nSuccess = SQLTables( nHandle [, cTableType [, cResultCursor ] ] )
```

The CursorAdapter Builder uses SQLTables() to populate a combobox with the list of tables when you specify an ODBC data source. Figure 4 shows the relevant form and Listing 15 shows the code for that case.



Account.*
Account.AccountNumber
Account.AccountType
Account.Balance
Account.BankNumber
Account.CashFlowClassification
Account.CurrencyRefFullName

Figure 4. This form from the CursorAdapter Builder, uses SQLTables() and SQLColumns() to retrieve data about an ODBC data source.

Listing 15. This code, from the GetTables method of the SelectCommandBuilderForm, populates a combo on the form with the list of tables in the remote database.

```
case vartype(.uConnection) = 'N'
  sqltables(.uConnection, 'TABLE', '_Tables')
  scan for upper(TABLE_NAME) <> 'DTPROPERTIES'
    lcTable = trim(TABLE_NAME)
    .AddTableToList(lcTable, lcTable)
  endscan for upper(TABLE_NAME) <> 'DTPROPERTIES'
  use in _Tables
```

SQLColumns() retrieves a list of the columns in a table. You can choose whether to format the list in the customary VFP way or in the format native to the server. Listing 16 shows the syntax for the function. Like SQLTables(), the return value can 1, 0 or a negative value.

Listing 16. SQLColumns can retrieve field information in the standard VFP format or the server's format.

```
nSuccess = SQLColumns(nHandle, cTable [, cFormat [, cResultCursor ] ] )
```

The CursorAdapter Builder uses SQLColumns() to collect the data for the mover in Figure 4. The code is shown in Listing 17.

Listing 17. This code, from the GetFieldsForTable method of the CursorAdapter Builder's SelectCommandBuilderForm, collects the list of fields used in the two-column mover in Figure 4.

```
case vartype(.uConnection) = 'N'
  * If we're using an ODBC connection handle, use SQLCOLUMNS to get the fields.
  * If it fails the first time, try again because sometimes it fails immediately
  * after using SQLCOLUMNS().
  sqlcolumns(.uConnection, lcTable, 'NATIVE', '_Fields')
  if not used('_Fields')
    sqlcolumns(.uConnection, lcTable, 'NATIVE', '_Fields')
  endif not used('_Fields')
  if used('_Fields')
    scan
      lcField  = trim(COLUMN_NAME)
      lcField  = GetObjectName(trim(COLUMN_NAME))
      lnFields = lnFields + 1
      dimension laFields[lnFields]
      laFields[lnFields] = lcTable + '.' + lcField
    endscan
  use in _Fields
endif used('_Fields')
```

Determining what's in use

Several functions let you find out what database and tables are open and what data sessions are in use. These are useful in several ways:

- Allowing you to save and restore the data set-up, if you need to change it in a tool;
- Letting you figure out what to operate on;
- Reporting the data set-up in an error handler.

ASessions() fills an array with a list of data sessions in use. It takes an array as its only parameter; the array comes back with a single column, listing the data session numbers in use. You might think this function is unnecessary, but it is possible for the data sessions in use to have non-sequential numbers. Suppose you open three forms, each with a private data session. At that point, you'd have data sessions 1 through 4 in use (1 is the default, shared, data session). If you then close the first form you opened, data session 2 is no longer is use; you have data sessions 1, 3 and 4.

ADatabases() fills an array with the list of open databases. It, too, takes a single parameter, the array name. The resulting array has two columns; the first tells you the name (just the filestem) of the database, while the second contains the path to the DBF.

AUsed() fills an array with a list of tables in use in the current or a specified data session; its syntax is shown in Listing 18. As you'd expect, omitting the second parameter applies the function to the current data session.

Listing 18. Call AUsed() to find out what tables are open in a particular data session.

```
nTables = AUSED( ArrayName [, nDataSession] )
```

The code in Listing 19 demonstrates both ADatabases() and AUsed(). It's part of the code to populate the combo and listboxes in the _tablemover class of the FFC (FoxPro Foundation Classes).

Listing 19. This code, from the InitData method of the _TableMover class, adds open databases to a combo box, and adds open tables to the list of tables.

```
m.nDBCCount=ADATABASES(aDBC)
FOR i = 1 TO m.nDBCCount
  * Add bar for popup
  IF m.i = 1
    THIS.cboData.AddItem("\-")
  ENDIF
  THIS.cboData.AddItem(aDBC[m.i,1])
ENDFOR

* Go thru workareas and see which tables open
m.nTotWorkAreas = AUSED(aWorkAreas)
FOR m.nCount = 1 TO m.nTotWorkAreas
  m.nWorkArea = aWorkAreas[m.nCount,2]

  * Avoid specific tables used by wizard and not in a DBC
  DO CASE
  CASE ASCAN(aSkipTables,DBF(m.nWorkArea))#0
    LOOP
  CASE ISREADONLY(m.nWorkArea) AND !THIS.AllowReadOnly
    * skip for read-only tables and queries
    LOOP
  CASE EMPTY(THIS.GetDBCName(m.nWorkArea))
    * Add to free tables list
    IF ATC(".TMP",DBF(m.nWorkArea))#0 AND !THIS.AllowQuery
      LOOP
    ENDIF
    IF !EMPTY(aDBFList[1])
      DIMENSION aDBFList[ALEN(aDBFList,1)+1,2]
    ENDIF
    aDBFList[ALEN(aDBFList,1),1] = DBF(m.nWorkArea)
    aDBFList[ALEN(aDBFList,1),2] = ALIAS(m.nWorkArea)
  OTHERWISE
    * Need to determine if its a Table, Local View or Remote View
    * Add to DBC tables list
    IF !THIS.AllowViews AND CURSORGETPROP("sourcetype",m.nWorkArea)#3
      LOOP
    ENDIF
    IF !EMPTY(aDBCList[1])
      DIMENSION aDBCList[ALEN(aDBCList,1)+1,2]
    ENDIF
    IF CURSORGETPROP("sourcetype",m.nWorkArea)#3  &&handle view here
      aDBCList[ALEN(aDBCList,1),1] = UPPER(CURSORGETPROP("sourcename",m.nWorkArea))
    ELSE
```

```
      aDBCList[ALEN(aDBCList,1),1] = DBF(m.nWorkArea)
    ENDIF
    aDBCList[ALEN(aDBCList,1),2] = ALIAS(m.nWorkArea)
  ENDCASE
ENDFOR
```

Exploring classes and forms

Developer tools often need to see what's inside class libraries, classes and forms. VFP has a strong set of tools for exploring and modifying the contents of class libraries and forms. (This section looks only at VCX-based classes. See "Working with code," later in this document for ways to peek inside PRG-based classes.)

This section looks at VFP language elements that let you find out what's in class libraries as well as examine and modify classes and forms. In addition to what's listed here, you can, of course, open any class library or form as a table and examine it that way; see "Treating VFP files as data," earlier in this document.

Looking into class libraries

Two functions let you look into 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.

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 20 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 20. 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
```

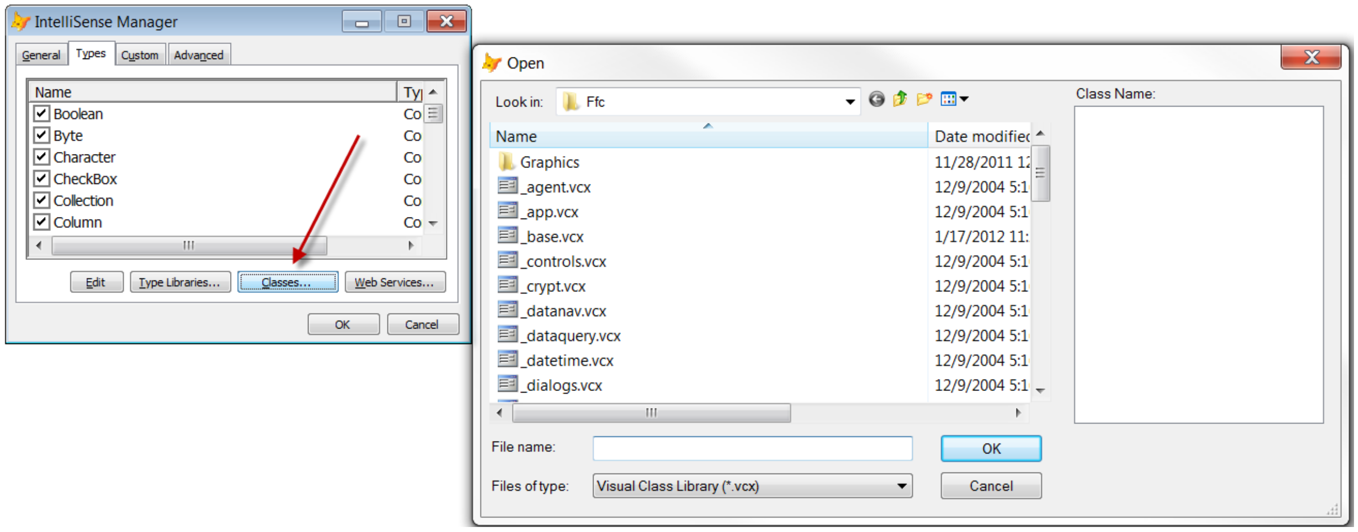
AGetClass() is useful for the user interface of developer tools. It displays the Open dialog, set up to select a class library, and puts information about the selected class library into a two-element array. You can pass a number of parameters to customize the dialog; Listing 21 shows the syntax.

Listing 21. 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 5. The relevant code is shown in Listing 22, though most of the method is omitted here.



Name	Date modified
Graphics	11/28/2011 12
_agent.vcx	12/9/2004 5:1
_app.vcx	12/9/2004 5:1
_base.vcx	1/17/2012 11:
_controls.vcx	12/9/2004 5:1
_crypt.vcx	12/9/2004 5:1
_datanav.vcx	12/9/2004 5:1
_dataquery.vcx	12/9/2004 5:1
_datetime.vcx	12/9/2004 5:1
_dialogs.vcx	12/9/2004 5:1

Figure 5. The Classes… button in the IntelliSense Manager calls AGetClass() to show the Open dialog, configured for choosing a class.

Listing 22. 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 handle for the object being designed, as well as for 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 23, while Table 3 shows the possible values for the second parameter.

Listing 23. ASelObj() lets you grab references to whatever is selected in the Form or Class Designer.

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

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.

ASelObj() is used widely in VFP tools. Listing 24 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.

Listing 24. This code, from the MemberDataEngine.SetupEngine method, 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
```

Two 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 25.

Listing 25. 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 too add the item to that category). The code in Listing 26 appears in the DropObject method of the _root class from which all Toolbox tools are derived.

Listing 26. 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 points 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 27 shows that part of the tool code.

Listing 27. 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 parameters you pass. Listing 28 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 28. 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 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.

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.

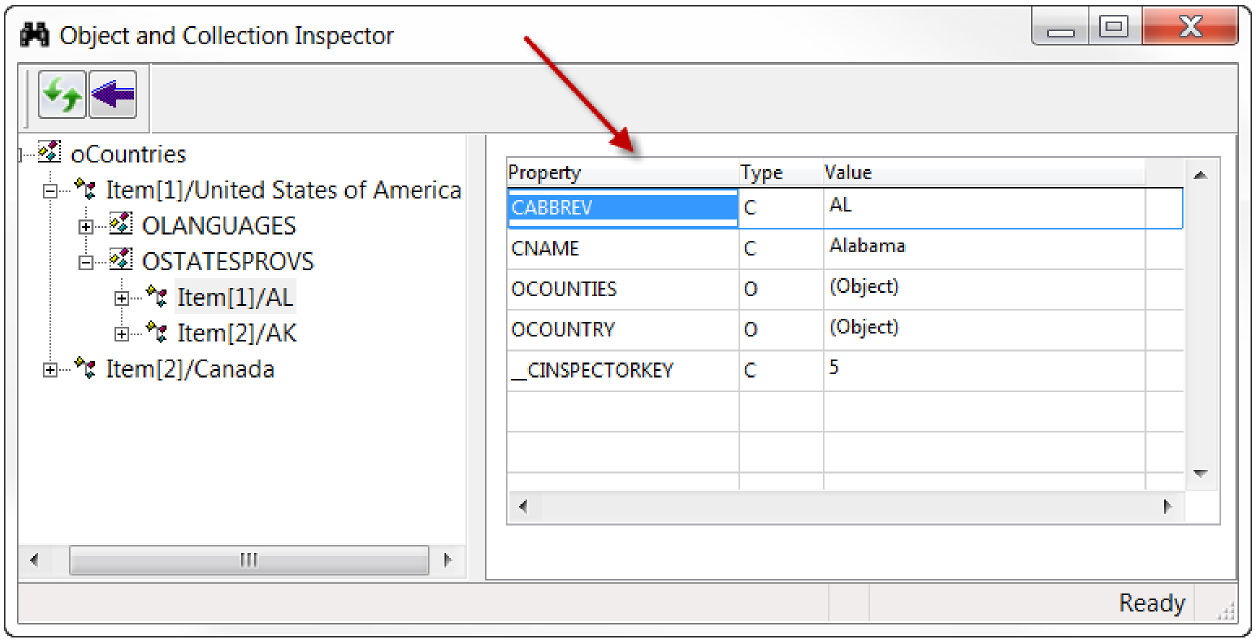
Flag Character	Group	Indicates
"G"	Visibility	Include public PEMs.
"H"	Visibility	Include hidden PEMs.
"P"	Visibility	Include protected PEMs.
"N"	Origin	Include native PEMs, that is, those that are part of the object's base class.
"U"	Origin	Include user-defined PEMs, those added at some point in the class hierarchy.
"B"	Inheritance	Include PEMs defined at this level (that is, not inherited).

Flag Character	Group	Indicates
“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.

AMembers() is extremely valuable when building any tool that needs to address the list of PEMs for an object. Listing 29 shows code from the Object Inspector I built 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 6 shows the tool in use, with an arrow indicating the grid.

Listing 29. This code from the Object Inspector put properties and their values into a grid for display.

```
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
```



Property	Type	Value
CABBREV	C	AL
CNAME	C	Alabama
OCOUNTIES	O	(Object)
OCOUNTRY	O	(Object)
_CINSPECTORKEY	C	5

Figure 6. The code in Listing 29 is used to create the cursor that populates the grid in this tool.

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 30 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().

Listing 30. 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 31, drawn from the Toolbox (specifically, the OnCompleteDrag method of the _ImageTool class) does both.

Listing 31. 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 you 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 32.

Listing 32. 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 33. (Note the use of PEMStatus() as well to ensure that the specified object actually has the method in question and has been changed.)

Listing 33. 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 34 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 34. Use WriteMethod to create methods and to populate them.

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

The code in Listing 35 comes from the PEMEditor_Utils class library of the PEM Editor; it's used when copying PEMs from one object to another.

Listing 35. 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 36.

Listing 36. 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 37, 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 37. 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
```

Working with code

Many developer tools need to work with code, whether displaying it for editing, or analyzing it in some way. VFP provides some functions that simplify the task. (It's also worth noting that VFP has a nice collection of functions for examining and manipulating text, such as SUBSTR(), AT(), STUFF(), and so forth. They're beyond the scope of this session. See Steven Black's excellent paper on working with text in VFP at http://stevenblack.com/text.html.)

What's in there?

Even though lots of code lives in forms and visual classes, chances are that your application still includes at least a few PRGs. In addition to the main program for a project, you likely have some standard functions stored in program files, and you may also have non-visual classes (such as the application class or various processing classes). In a couple of projects that I'm maintaining, there are procedure files, as well.

A number of the tools that come with VFP use PRGs. For example, the heart of the Toolbox is the ToolboxEngine class stored in ToolboxEngine.PRG.

VFP provides some tools for digging into programmatic code. The most important is the AProcInfo() function, which takes a filename and creates an array of information about the file contents. Listing 38 shows the syntax. When you omit the third parameter (or pass 0), the array contains an entry for each item in the file: procedures/functions, classes, methods and compiler directives. You can pass 1, 2 or 3 to restrict the listing to classes, methods, or compiler directives respectively. Unfortunately, there's no way to get information about just procedures and functions.

Listing 38. The AProcInfo() function puts information about the contents of a program file into an array.

```
nCount = AProcInfo( aResult, cFileName [, nItemType] )
```

The resulting array contains the name of each item and the line number where it's found in the file. Depending on the value of nItemType, there may be additional columns. Table 7 shows the possibilities.

Table 7. The columns in the array created by AProcInfo() vary based on the value passed for nItemType.

Column	Contents	Applies for nItemType
1	Name of item. For methods, the name includes the class name. For classes when nItemType=0, includes the "AS" portion of the definition listing the parent class.	0,2,3
	Class name.	1
2	Line number in the file where the item occurs.	All
3	Item type: Define, Directive, Class, Procedure. All procedures, functions, methods and events are classified as "Procedure."	0,3
	Parent class name.	1
4	Indentation, but always 0.	0
	If the class is defined OLEPublic, "OLEPUBLIC". Otherwise, empty.	1

Several of the tools that come with VFP use AProcInfo(). For example, the Code References tool uses it in a method that determines in what method of a visual class a particular line number occurs. The code for that method, called GetProcedure, is shown in Listing 39.

After using AProcInfo() to find the starting position of each item in the file, the code loops through the array and compares the start position of the item to the line it's looking for. As the loop proceeds, each Procedure type line (which includes procedures, functions and methods) is parsed and the name of the method saved. Then, when the loop reaches an item that's beyond the line it's looking for, the saved procedure name is returned.

Listing 39. This method uses AProcInfo() to find out what procedure or method in a file contains a specified line.

```
FUNCTION GetProcedure(nLineNo)
  LOCAL cTempFilename
  LOCAL cSafety
  LOCAL nCnt
  LOCAL i
  LOCAL cProcName
  LOCAL ARRAY aFileInfo[1]

  IF m.nLineNo == 0 OR EMPTY(THIS.cFileText)
    RETURN ''
  ENDIF
  m.cProcName = ''
  m.cTempFilename = ADDBS(SYS(2023)) + SYS(2015) + ".tmp"

  TRY
    STRTOFILE(THIS.cFileText, m.cTempFilename, 0)
    m.nCnt = APROCINFO(aFileInfo, m.cTempFilename, 0)

  CATCH
    m.nCnt = 0
  ENDTRY
  FOR m.i = 1 TO nCnt
    IF m.nLineNo <= aFileInfo[m.i, 2]
      EXIT
    ENDIF

    DO CASE
    CASE aFileInfo[m.i, 3] == "Procedure"
      m.cProcName = aFileInfo[m.i, 1]
      IF AT('.', m.cProcName) > 0
        m.cProcName = SUBSTRC(m.cProcName, AT('.', m.cProcName) + 1)
      ENDIF
     CASE aFileInfo[m.i, 3] == "Class"
      m.cProcName = ''
    ENDCASE
  ENDFOR

  m.cSafety = SET("SAFETY")
  SET SAFETY OFF
  ERASE (m.cTempFilename)
  SET SAFETY &cSafety

  RETURN m.cProcName
ENDFUNC
```

Opening code windows

VFP's EditSource() function lets you open pretty much any kind of file that contains code in the appropriate editor. The parameters to pass depend on the file type, but always include the file name. (Actually, there's a way to use this function without passing the file name, but it's not relevant to this discussion.) If you also pass a line number within the file, the appropriate editor opens with the cursor positioned on the specified line. Listing 40 shows the syntax.

Listing 40. The EditSource() function is your one-stop technique for opening code windows.

```
nErrorCode = EditSource( cFileName [, nLineNo [, cClassName [, cMethodName ] ] ] )
```

The cClassName and cMethodName parameters apply only to visual classes and forms. As you'd expect, when you include them, the specified class opens to the specified method.

One of the cool things about this function is that it works for almost every VFP file type you want to open. Pass it an FRX and it opens the Report Designer. Pass an MNX and it opens the Menu Designer. Pass a DBC and it opens the code editor for the stored procedures of that database. (However, if you pass it a file type it doesn't know what to do with, such a project or table, it opens the file as a text file.)

A number of VFP's XBase and Thor tools use EditSource() to open files. Listing 41 shows a method from Code References that uses EditSource() to open the file the user is looking for.

Listing 41. This method from the Code References tool uses EditSource() to open the right file.

```
FUNCTION GotoReference(cUniqueID)
  LOCAL nSelect
  LOCAL cFilename
  LOCAL cFileType
  LOCAL cClassName
  LOCAL cProcName

  IF VARTYPE(cUniqueID) <> 'C' OR EMPTY(cUniqueID)
    RETURN .F.
  ENDIF

  IF USED("FoxRefCursor") AND SEEK(cUniqueID, "FoxRefCursor", "UniqueID")
    nSelect = SELECT()

    cFilename  = ADDBS(RTRIM(FoxRefCursor.Folder)) + RTRIM(FoxRefCursor.FileName)
    cClassName = RTRIM(FoxRefCursor.ClassName)
    cProcName  = RTRIM(FoxRefCursor.ProcName)
    cFileType  = UPPER(JUSTEXT(cFileName))

    DO CASE
    CASE cFileType == "SCX"
      EDITSOURCE(cFileName, MAX(FoxRefCursor.ProcLineNo, 1), cClassName, cProcName)

    CASE cFileType == "VCX"
      EDITSOURCE(cFileName, MAX(FoxRefCursor.ProcLineNo, 1), cClassName, cProcName)

    CASE cFileType == "DBF"
      * do a TRY/CATCH here
      IF USED(JUSTSTEM(cFilename))
        SELECT (JUSTSTEM(cFilename))
      ELSE
        SELECT 0
        USE (cFilename) EXCLUSIVE
      ENDIF
      MODIFY STRUCTURE

    OTHERWISE
      EDITSOURCE(cFileName, FoxRefCursor.LineNo)
    ENDCASE

    SELECT (nSelect)
  ENDIF

ENDFUNC
```

It’s also worth noting that VFP’s various MODIFY commands can be used programmatically. Just be sure to include the NOWAIT keyword if you don’t want your code to stop and wait until you close the editing window. The advantage of EditSource() is that you don’t have to figure out which editor to use.

A number of the Xbase tools use the various MODIFY commands. For example, the code in Listing 42 comes from the MemberData Editor. It’s the Click method of the View XML button, which displays the complete MemberData for the class or form.

Listing 42. This code, in the Click method of the View XML button of the MemberData Editor, saves the XML to a file, then uses MODIFY FILE to display it, and then erases the file.

```
* Display the MemberData XML.

local lcXML, ;
  lcFile
with Thisform
  if .nScope = cnSCOPE_OBJECT
    lcXML = .GetMemberDataXML()
  else
    lcXML = .GetMemberDataForContainer(.cSelectedParent, .T.)
  endif .nScope = cnSCOPE_OBJECT
  do case
    case empty(lcXML)
      messagebox(ccLOC_NO_MEMBERDATA, MB_OK + MB_ICONINFORMATION, ;
        .Caption)
    case not isnull(lcXML)
      lcFile = addbs(sys(2023)) + '_MemberData.XML'
      strtofile(lcXML, lcFile)
      modify file (lcFile) noedit
      erase (lcFile)
  endcase
endwith
```

Processing projects

As noted in “Treating VFP files as data,” earlier in this document, VFP’s projects are just tables with a special extension. So you can process project data by opening the project with USE, and treating it like a table.

However, projects offer another way to explore, as well as a unique tool that lets you respond to actions on a project.

The Project and File objects

For each project open in the Project Manager, there is a corresponding object based on the Project class. The Project class includes a Files property, which references a collection of File objects. (The Project and File objects are COM classes, not native VFP classes, which affects some of what you can with them.) The _VFP object that references VFP’s engine has a Projects property that references a collection of open projects, and an ActiveProject property that references the current project.

You can use these objects and properties to access and modify projects. For example, the code in Listing 43 comes from a method called GoToDefFindClassInPath that's part of the Thor Go To Definition tool. It loops through the files in the active project twice. The first time, it finds all class libraries (VCX files) and checks each for a specified class. If the class isn't found in a VCX, the second loop looks for program (PRG) files and checks each of those for the specified class.

Listing 43. This code from the Thor Go To Definition tool looks for a specified class in a project.

```
If Not llFound And Type ('_vfp.ActiveProject') = 'O'
  For Each loFile In _vfp.ActiveProject.Files
    If Not llFound And loFile.Type = 'V'
      This.GoToDefProcessVCXForClass (loFile.Name, tcName, toInclude)
      If Not Empty (toInclude.File)
        llFound = .T.
        Exit
      Endif Not Empty (toInclude.File)
    Endif loFile.Type = 'P'
  Next loFile

  * Check all PRGs in the active project.

  For Each loFile In _vfp.ActiveProject.Files
    If Not llFound And loFile.Type = 'P'
      lcPRG = loFile.Name
      This.GoToDefProcessPRGForClass (lcPRG, tcName, toInclude, ;
                                      Upper (Filetostr (lcPRG)))
      If Not Empty (toInclude.File)
        llFound = .T.
        Exit
      Endif Not Empty (toInclude.File)
    Endif loFile.Type = 'P'
  Next loFile
Endif Not llFound ...
```

I've written a lot of utilities that work with the Project and File objects. In fact, I created a class of project utilities (included in the downloads for this session) to help me understand and clean up issues in one project I inherited. For example, the method in Listing 44 fills a cursor with a list of all files that are referenced in the project but can't be found.

Listing 44. This method from a class of tools for working with projects puts a list of missing files into a cursor.

```
PROCEDURE ListMissingFiles(cAlias)
* Create a list of files in the project that are not
* found in the project folders.

LOCAL oFile, cFilewithPath

cAlias = This.GetValidAlias(m.cAlias, "csrMissingFiles")

CREATE CURSOR (m.cAlias) ;
  (iID I AUTOINC, mFileName M)
```

```
FOR EACH oFile IN This.oProject.Files
  cFileWithPath = oFile.Name
  IF NOT FILE(m.cFilewithPath)
    INSERT INTO (m.cAlias) (mFileName) VALUES (m.cFileWithPath)
  ENDIF
ENDFOR

RETURN m.cAlias
```

You're not limited to just looking at the contents of projects; you can modify them. Listing 45 shows some of the code from another method in the same class. The CopyProject method makes a copy of an existing project in a new location, copying the files that are actually referenced in the project. Note the use of the Add method of the Files collection near the end of the listing. (Additional code not shown here copies the project's icon, and checks forms and classes for any graphics files that haven't yet been copied.)

Listing 45. This method (shown only in part) makes a copy of a project in a specified folder, copying the files in the project.

```
PROCEDURE CopyProject(cNewRoot)
* Move a project and all the included files
* from the current folder to the specified
* root, maintaining the current folder structure.

LOCAL oFile, cNewName, cNewPath, cNewProject, oNewProject
LOCAL cMissing, cNoAdd, cFileExt, cNewExt

IF PCOUNT() < 1 OR EMPTY(m.cNewRoot)
  MESSAGEBOX("CopyProject: You must specify the folder for the copy.")
  RETURN
ENDIF

* First, does the new folder exist
IF NOT DIRECTORY(m.cNewRoot)
  MD (m.cNewRoot)
ENDIF

* Create the new project.
This.ReportToUser("Creating new project")

cNewProject = FORCEPATH(This.oProject.Name, m.cNewRoot)
CREATE PROJECT (m.cNewProject) NOWAIT
oNewProject = _VFP.ActiveProject

* Modified 22-April-2011 by TEG
* Need to set HomeDir explicitly
oNewProject.HomeDir = m.cNewRoot

* Modified 13-April-2011 by TEG
* Bring project properties along
WITH oNewProject
  .VersionComments = This.oProject.VersionComments
  .VersionCompany = This.oProject.VersionCompany
  .VersionCopyright = This.oProject.VersionCopyright
  .VersionDescription = This.oProject.VersionDescription
  .VersionLanguage = This.oProject.VersionLanguage
  .VersionNumber = This.oProject.VersionNumber
  .VersionProduct = This.oProject.VersionProduct
  .VersionTrademarks = This.oProject.VersionTrademarks

  .Encrypted = This.oProject.Encrypted
ENDWITH

* Copy all files to appropriate directories
cMissing = ""
cNoAdd = ""
FOR EACH oFile IN This.oProject.Files
  This.ReportToUser("Handling file " + oFile.Name)

  IF NOT FILE(oFile.Name)
    * Original file is missing. Make a list for user.
    cMissing = m.cMissing + CHR(13) + CHR(10) + oFile.Name
    LOOP
  ENDIF

  IF This.cHomeDir $ oFile.Name
    cNewName = m.cNewRoot + STREXTRACT(oFile.Name, This.cHomeDir,"",1,3)
    cNewFilePath = JUSTPATH(m.cNewName)
    IF NOT FILE(m.cNewName)
      IF NOT DIRECTORY(m.cNewFilePath)
        MD (m.cNewFilePath)
      ENDIF
      COPY FILE (oFile.Name) TO (m.cNewName)
    ENDIF

    * Copy associated memo file.
    cFileExt = JUSTEXT(m.cNewName)
    IF INLIST(UPPER(m.cFileExt), "SCX", "VCX", "MNX", "FRX", "LBX")
      cNewExt = LEFT(m.cFileExt,2) + "T"
      COPY FILE (FORCEEXT(oFile.Name, m.cNewExt)) TO ;
                (FORCEEXT(m.cNewName, m.cNewExt))
    ENDIF
  ELSE
    * If the original file is not in the project
    * folder hierarchy, don't copy it. Just add the
    * original to the new project.
    cNewName = oFile.Name
  ENDIF

  * Now add it to the new project. Wrap in TRY-CATCH
  * in case it's already there.
  TRY
    oNewFile = oNewProject.Files.Add(m.cNewName)

    * Check whether it's the main file in the old project, and,
    * if so, make it the main here.
    IF UPPER(oFile.Name) == UPPER(This.oProject.MainFile)
      oNewProject.SetMain(m.cNewName)
    ENDIF
  CATCH
    cNoAdd = m.cNoAdd + CHR(13) + CHR(10) + m.cNewName
  ENDTRY

ENDFOR
```

ProjectHooks

In addition to the ability to address projects and their contents directly, VFP has a class called ProjectHook that essentially provides you with project events. You can attach a ProjectHook class to any project. Once you've done so (and closed and reopened the project), the projecthook's events fire when the corresponding actions take place on the project. Many of the projecthook events correspond directly to a method of the Project object. Table 8 shows the available events.

Table 8. ProjectHooks give a project a set of events that fire in response to actions on the project.

Event	Fires
Activate	When the project gets focus. That is, when the instance of the Project Manager for this project is activated.
AfterBuild	When a build is finished. Receives a parameter indicating whether an error prevented the build from completing. (0 indicates success.)
BeforeBuild	Before a build begins. Allows you to make changes before doing the build. Receives the same parameters as the project's Build method, and can change them before they're passed on to Build.
Deactivate	When the project loses focus.
Destroy	When the project is closed.
Error	When an error occurs in the projecthook's code.
Init	When the project is opened.
OLEDragDrop	When an OLE drag-and-drop operation ends by dropping onto the Project Manager.
OLEDragOver	When an item in an OLE drag-and-drop is dragged over the Project Manager.
QueryAddFile	When a file is added to the project. Can prevent the file from being added.
QueryModifyFile	When a file is opened for editing. Can prevent the file from being opened.
QueryNewFile	When a new file is being created. Can prevent the file from being created.
QueryRemoveFile	When a file is removed from the project. Can prevent the file from being removed.
QueryRunFile	When a file is executed. Can prevent execution of the file.
SCCDestroy	Before the Destroy event to provide an opportunity to take source control actions.
SCCInit	After the Init event to provide an opportunity to take source control actions.

A number of these events can prevent the corresponding project action. Just include NODEFAULT in their code. For example, putting NODEFAULT in BeforeBuild prevents the project from being built. NODEFAULT in any of the QueryXXXFile events prevents the file action from taking place.

Oddly, projecthooks don't automatically get a reference to the project with which they're associated, so it's a best practice to add a property to the projecthook for that purpose and set it in the Init method, as in Listing 46.

Listing 46. Setting a custom property of a projecthook to the associated project is a best practice. This code in Init handles it.

```
* Assumes you've added a property called oProject to the class.
```

```
This.oProject = _VFP.ActiveProject
```

There are very few examples of projecthooks in the code that comes with VFP or in VFPX. I suspect that's because what you want in a projecthook depends a lot on the way you work. VFPX offers ProjectHookX, a projecthook designed to make it possible to attach multiple projecthooks to a single project.

Rick Schummer uses projecthooks extensively. His free WLC ProjectHook (available at http://www.whitelightcomputing.com/prodprojectbuilder.htm) offers a wide variety of behaviors, including an alternative approach to mixing and matching different behaviors. The core class is phkDevelopment, which is the one you actually attach to a project. Listing 47 shows the code in its Activate method, while Listing 48 shows the code in the custom ChangeToProjectDirectory method.

Listing 47. This code, in the Activate method of a projecthook, switches to the project's home directory whenever the project becomes active. It also sets up a custom project toolbar.

```
* Change the directory to the project's home directory
this.ChangeToProjectDirectory()

DODEFAULT()

* RAS 28-Jan-2000 Added the ProjectTools Toolbar
this.CreateProjectToolbar()

RETURN
```

Listing 48. The ChangeToProjectDirectory method of the WLC projecthook class is called from the Activate method. It makes the project's home directory current, and then allows any extension objects hooked into the projecthook to run their code for this method.

```
* Set the default directory to the project's home
* directory so the generic pathing works
*  ie SET PATH TO data, forms, classes, graphics
IF TYPE("this.oProject") = "O" AND !ISNULL(this.oProject)
   IF !EMPTY(this.oProject.HomeDir)
      CD (this.oProject.HomeDir)
   ELSE
      this.DeveloperMessage("Project directory setting is empty...", .T.)
   ENDIF
ELSE
   * This should never happen, unless you manually
   * CREATEOBJECT() the class without a project.
   THIS.DeveloperMessage("Project reference not available", .T.)
ENDIF

* Hook into additional code provided in extension object(s).
FOR lnIndex = 1 TO this.nHooks
   IF NOT ISNULL(this.aHook[lnIndex, 1])
      llHookMethod = PEMSTATUS(this.aHook[lnIndex, 1], "ChangeToProjectDirectory", 5)

      IF llHookMethod
         this.aHook[lnIndex, 1].ChangeToProjectDirectory()
      ENDIF
   ENDIF
ENDFOR

RETURN
```

Summing up

VFP provides a rich set of language elements for building developer tools. Using just native language elements, you can explore data, forms, classes, programs, menus, reports, labels and projects. In addition, VFP comes with the source code for a number of the included developer tools, which gives you a rich set of examples to explore when you're designing a new tool.