VFP: Ideal for Tools, Part 3

The VFP language supports programmatic manipulation of programs and projects, providing more options for creating developer tools.

Tamar E. Granor, Ph.D.

The first two parts of this series looked at Visual FoxPro language elements for exploring data, and for working with classes and forms. This article looks at parts of VFP that let you work with programs and projects.

As with the other articles, most of the examples here are drawn from code that comes with VFP or is included in VFPX.

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 article. See Steven Black's excellent paper on working with text in VFP at http://stevenblack.com/articles/text-and-string-handling-in-VFP/.)

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 1 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 1. 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 1 shows the possibilities.

Table 1. 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 2.

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 2. 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 3 shows the syntax.

Listing 3. 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 4 shows a method from Code References that uses EditSource() to open the file the user is looking for.

Listing 4. 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 5 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 5. 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 my last article, 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 them, 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 6 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 6. 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 (originally published in the May, 2011 issue, and included in the downloads for this article) to help me understand and clean up issues in one project I inherited. For example, the method in Listing 7 fills a cursor with a list of all files that are referenced in the project but can't be found.

Listing 7. 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 8 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 8. 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,
LOCAL 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 2 shows the available events.

Table 2. 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 9.

Listing 9. 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 10** shows the code in its Activate method, while **Listing 11** shows the code in the custom ChangeToProjectDirectory method.

Listing 10. 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()

this.CreateProjectToolbar()

RETURN
```

Listing 11. 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.

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