Build Your Own Project Tools

VFP makes it easy to write code to explore and manipulate projects and their contents.

Tamar E. Granor, Ph.D.

While Visual FoxPro's Project Manager doesn't offer much in the way of tools to audit or manage projects, the ability to address the project as an object more than makes up for this deficiency.

Much of my work is with projects originally written by someone else. Usually, in those cases, I'm taking over for the original developer. However, I recently started helping an existing team of developers who are moving a project forward.

As I was examining the project, trying to get a feel for how it works, I found a lot of extra stuff in the project folders. I also noticed that some files didn't seem to be where they should. In other words, as tends to happen over time, the project had gotten messy.

When I take over a project, I have no qualms about simply creating a new PJX, adding the main program and clicking Build to see what really belongs in the project. I'll generally throw my new project and its associated files into a new folder, and work from there, thus leaving the mess behind.

But in this case, I couldn't start moving things around; others were using them. But I still needed to get a handle on what was in use and what wasn't. Fortunately, VFP gave me the tools to do so with just a little code. I ended up creating a class that's really a suite of project exploration and management tools.

The Project and File Objects

Since VFP 5, every open project is associated with a Project object based on a COM class. Each Project is a member of a Projects collection. You can access both the collection and the active project (the one with focus) using the _VFP system variable. _VFP.Projects addresses the entire collection, while _VFP.ActiveProject gives you a reference to the active project. As soon as you open a project, it becomes _VFP.ActiveProject; that reference remains constant until you close the project, or open or click on another project.

Project has a number of useful properties, such as Name, which contains the fully-qualified path and filename for the project; and HomeDir, the home directory of the project. It also has a Files collection that provides access to each file in the project. Files contains File objects, also COM-based. The File object has properties including Name, the fully-qualified path and filename; and Type, a single-character indicating the type of file.

You can use these objects to explore a project and its contents. Listing 1 shows a simple example. It runs through the files in the active project and sends their names to the Debug Output window. To try it, just open a project, open the Debugger, and run the code.

Listing 1. Walking through a project is easy with the Project object and its Files collection.

```
LOCAL oFile

FOR EACH oFile IN _VFP.ActiveProject.Files
  DEBUGOUT oFile.Name
ENDFOR
```

Getting Started

For my project tools, I clearly needed to hold onto a reference to the relevant project. As I started, I suspected there'd be other information I'd want to keep at hand, so I created a class based on the Custom class, and added custom properties oProject and cHomeDir to hold a reference to the project and the project's home directory, respectively.

The GetProject method allows the user to specify the project to work with. It opens the specified project, and sets the oProject and cHomeDir properties. Listing 2 shows the code. Note the NOWAIT clause on MODIFY PROJECT to keep the project open. You might also want to add the NOSHOW clause so that the Project Manager isn't visible.

Listing 2. The GetProject method accepts the name of a project, opens it and stores a reference to it.

```
PROCEDURE GetProject(cProject)
* Connect a project to this object.

LOCAL lProceed

IF VARTYPE(m.cProject) = "C"
  TRY
    MODIFY PROJECT (m.cProject) NOWAIT
    This.oProject = _VFP.ActiveProject
    lProceed = .t.
  CATCH
    lProceed = .F.
```

```
  ENDTRY
ELSE
  lProceed = .F.
ENDIF

IF NOT m.lProceed
  MESSAGEBOX("Unable to open specified " + ;
             "project: " + m.cProject + ;
             ". Make sure to provide a " + ;
             "good path.")
  RETURN
ENDIF

This.cHomeDir = This.oProject.HomeDir
RETURN
```

Indentifying Unused Files

My first priority when writing this class was getting a list of files in the project folders that weren't actually in the project. ListUnusedFiles looks through the folder containing the project and all its subfolders and makes a list of all the files that are not in the project. It allows you to specify a comma-separated list of folders to exclude from the search. That's useful if, like me, you have certain folders you use for items that are never added to the project.

While it's possible to check each file name directly against the project to see whether it exists, given VFP's data processing speed, it made more sense to me to fill a cursor with the list of files in the project. Then, I could use SEEK to check whether a file is in the project. Listing 3 shows the BuildFileCursor method. It's not that different from the code in Listing 1; it loops through the Files collection and puts the file name and path for each into a cursor.

Listing 3. The BuildFileCursor method fills a cursor with the names of files in the project.

```
PROCEDURE BuildFileCursor(cAlias, lIndex)
* Build a cursor of files in the project.
* If indicated, index it on the path + file
* combination.

cAlias = This.GetValidAlias(m.cAlias, ;
  "csrProjectFiles")

CREATE CURSOR (m.cAlias) ;
  (iID I AUTOINC, mFile M, mPath M)
INDEX on iID TAG iID
IF m.lIndex
  INDEX ON UPPER(CAST(FORCEPATH(mFile, mPath);
     AS C(200))) TAG FilePath
ENDIF

LOCAL oFile, cFilename, cPath

FOR EACH oFile IN This.oProject.Files
  cFileName = JUSTFNAME(oFile.Name)
  cPath = JUSTPATH(oFile.Name)
  INSERT INTO (m.cAlias) (mFile, mPath) ;
    VALUES (m.cFileName, m.cPath)
ENDFOR

RETURN
```

The GetValidAlias method deserves a mention before proceeding. As this set of tools grew, I created more and more methods that put their results into a cursor. I wanted the user of the tool to be able to specify the alias for each cursor, but wanted to have a default alias if the user chose not to specify one.

I found myself writing the same code over and over to test the alias parameter and substitute a default. So I created GetValidAlias. It accepts two parameters, the proposed alias and a default to use if the proposed alias is missing or invalid. (Think of it as NVL() or EVL() for aliases.) Given its general utility, I think it's likely to wind up becoming a function in my toolkit, rather than leaving it as a method of this one class.

Identifying files in the project folders that aren't in the project is a recursive task. We want to start in the home directory, and check each folder it contains, and the folders they contain and so forth. Generally, recursive tasks call for at least two methods: one to set things up and kick the job off, and one to do the actual work. In this case, ListUnusedFiles (Listing 4) does the set-up work and ListUnusedFilesInOneFolder (Listing 5) is the recursive worker method.

Listing 4. The ListUnusedFiles method sets things up to look for files in the project folders that are not included in the project. Then, it gets the ball rolling by calling the recursive ListUnusedFilesInOneFolder.

```
PROCEDURE ListUnusedFiles( ;
  cAlias, cExcludeFolders)

IF VARTYPE(m.cExcludeFolders) <> "C"
  cExcludeFolders = ""
ELSE
  * Bracket with commas, so we can search for
  * the specified folder with commas on both
  * ends. Change to upper to simplify
  * searching. Also, prefix each with the
  * project's home directory.
  cExcludeFolders = "," + This.cHomeDir + ;
    "\" + UPPER(ALLTRIM( ;
    STRTRAN(m.cExcludeFolders, ",", ;
     "," + This.cHomeDir + "\"))) ;
    + ","
ENDIF

cAlias = This.GetValidAlias(m.cAlias, ;
  "csrUnusedFiles")

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

* First, build a cursor of files in the
* project.
This.BuildFileCursor("csrProjectFiles", .T.)

This.ListUnusedFilesInOneFolder(;
  This.cHomeDir, m.cAlias, ;
  "csrProjectFiles", m.cExcludeFolders)

RETURN m.cAlias
```

The trickiest issue in all of this is handling the excluded folders. I wanted to make it easy for the user to specify the folders to exclude, which means

not having to spell out the full path for each. So, the folders can be listed relative to the project's home directory. ListUnusedFolders prefixes each with the home directory. It also converts them all to upper-case and puts commas at both ends of the list, so that searching for a specific folder is easy.

Listing 5. The workhorse method for identifying unused files in project folders is ListUnusedFilesInOneFolder, which works its way through the folders recursively.

```
PROCEDURE ListUnusedFilesInOneFolder( ;
  cFolder, cResultAlias, ;
  cProjFilesAlias, cExcludeFolders)

LOCAL aFolderFiles[1], nFileCount, nFile, ;
    cFileName, cSubFolder

nFileCount = ADIR(aFolderFiles, ;
  FORCEPATH("*.*", m.cFolder), "D")

FOR nFile = 1 TO m.nFileCount
  cFileName = aFolderFiles[m.nFile, 1]
  IF NOT INLIST(cFileName, ".", "..")
    IF "D" $ aFolderFiles[m.nFile, 5]
      * It's a folder, so call this method
      * recursively unless it's excluded
      cSubfolder = UPPER(ALLTRIM( ;
        FORCEPATH(m.cFileName, m.cFolder)))
      IF NOT "," + m.cSubFolder + "," $ ;
        m.cExcludeFolders
        This.ListUnusedFilesInOneFolder( ;
          m.cSubFolder, m.cResultAlias, ;
          m.cProjFilesAlias, ;
          m.cExcludeFolders)
      ENDIF
    ELSE
      * Check for this file.
      cExt = JUSTEXT(m.cFileName)
      IF INLIST(m.cExt, "SCT", "VCT", ;
        "MNT", "FRT", "LBT")
        cAssocExt = LEFT(m.cExt, 2) + "X"
        cFileToSearch = ;
          FORCEEXT(JUSTSTEM(m.cFileName),;
           m.cAssocExt)
      ELSE
        cFileToSearch = m.cFileName
      ENDIF

      cFileToSearch = FORCEPATH( ;
        m.cFileToSearch, m.cFolder)
      IF NOT SEEK(m.cFileToSearch, ;
        m.cProjFilesAlias, "FilePath")
        INSERT INTO (m.cResultAlias) ;
          (mFileWithPath) ;
          VALUES ;
          (FORCEPATH(m.cFileName, ;
           m.cFolder))
      ENDIF
    ENDIF
  ENDIF
ENDFOR

RETURN RECCOUNT(m.cResultAlias)
```

The method is fairly straightforward. It uses ADIR() to get a list of all files and folders in the specified folder. Then it loops through that list. If the item is a folder, it checks it against the list of excluded folders. If the folder isn't excluded, the method calls itself recursively.

VFP stores many project elements in tables, using special extensions to indicate the type. (For example, form files use SCX for the DBF and SCT for the FPT.) The list of files in a project includes only the DBF portion of such files, not the memo file. So the method checks the extension of the file it's working on. If it's one of the special memo files, we search instead for the associated table file. If the specified file isn't found in the cursor of files in the project, we add it to the result cursor.

Find Files in the Home Directory

My next concern was identifying files stored in the project's home directory. It was clear to me that the intent on this project was to put all project files into subdirectories based on their type, but the home directory contained dozens of files. So I wanted to know which of them actually belonged to the project, and ultimately get them where they should be.

Finding these files was a much simpler task that identifying unused files. It required just a loop through the files in the project. ListFilesInRoot, shown in Listing 6, stores the list in a cursor.

Listing 6. ListFilesInRoot builds a list of files included in the project, but stored in the project's home directory.

```
PROCEDURE ListFilesInRoot(cAlias)

LOCAL oFile

cAlias = This.GetValidAlias( ;
  m.cAlias, "csrFilesInRoot")

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

LOCAL cUpperHome, cFileName

cUpperHome = UPPER(This.cHomeDir)

FOR EACH oFile IN This.oProject.Files
  cFileName = UPPER(oFile.Name)
  IF JUSTPATH(m.cFileName) == m.cUpperHome
    INSERT INTO (m.cAlias) (mFileName) ;
      VALUES (oFile.Name)
  ENDIF
ENDFOR

RETURN m.cAlias
```

Putting the files where they belong is a more complicated task. There are two tricky parts. The first is knowing where to put the files. To handle that, I added a set of properties to the class to hold the names of the folders for different project components. These properties have names like cFormDir and cClassLibDir. The MoveFilesFromRoot method (Listing 7) prompts the user to specify the right folder for each one the first time it's needed, and stores the result in the property.

Listing 7. Moving files from the project's home directory to the right subfolder can be tricky.

```
PROCEDURE MoveFilesFromRoot
```

```
LOCAL nFilesToMove, nFile, cFile, cPartnerFile
LOCAL cDestFolder, oFile, cType, nSkipped
LOCAL cDescription

nFilesToMove = ;
  This.ListFilesInRoot("csrFilesInRoot")

* Add a field to track whether we've
* successfully moved the file.
ALTER table csrFilesInRoot ADD lMoved L

nSkipped = 0

SELECT csrFilesInRoot
SCAN
  cFile = csrFilesInRoot.mFileName
  IF FILE(m.cFile)
    oFile = This.oProject.Files[m.cFile]
  ELSE
    nSkipped = m.nSkipped + 1
    LOOP
  ENDIF

  * Strategy is to collect all information,
  * then move the file, remove it from the
  * project, then add it back.

  * First, figure out what kind of file we
  * have, thus, where it's going and whether
  * there's a "partner" file.

  cType = UPPER(oFile.Type)
  cDestFolder = ""

  DO CASE
  CASE m.cType = "K"
    IF EMPTY(This.cFormDir)
      This.cFormDir = GETDIR("", ;
        "Specify form folder for " + ;
        "project","Form folder")
    ENDIF

    IF NOT EMPTY(This.cFormDir)
      cDestFolder = This.cFormDir
      cPartnerFile = FORCEEXT(m.cFile, "SCT")

      * Need to fix paths in the file
      This.FixClassLoc (m.cFile,;
        This.cHomeDir, This.cFormDir)
      This.FixPictures (m.cFile,;
        This.cHomeDir, This.cFormDir)
    ENDIF

  CASE m.cType = "V"
    IF EMPTY(This.cClassLibDir)
      This.cClassLibDir = GETDIR("", ;
        "Specify classlib folder for " + ;
        "project","Classlib folder")
    ENDIF

    IF NOT EMPTY(This.cClassLibDir)
      cDestFolder = This.cClassLibDir
      cPartnerFile = FORCEEXT(m.cFile, "VCT")

      * Need to fix paths in the file
      This.FixClassLoc(m.cFile, ;
        This.cHomeDir, This.cFormDir)
      This.FixPictures (m.cFile,;
        This.cHomeDir, This.cFormDir)
    ENDIF

  CASE m.cType = "P"
    IF EMPTY(This.cProgDir)
      This.cProgDir = GETDIR("", ;
        "Specify program folder for " +  ;
        "project","Program folder")
    ENDIF

    IF NOT EMPTY(This.cProgDir)
      cDestFolder = This.cProgDir
      cPartnerFile = ""
    ENDIF

  CASE m.cType = "M"
    IF EMPTY(This.cMenuDir)
      This.cMenuDir = GETDIR("",;
        "Specify menu folder for " + ;
        "project","Menu folder")
    ENDIF

    IF NOT EMPTY(This.cMenuDir)
      cDestFolder = This.cMenuDir
      cPartnerFile = FORCEEXT(m.cFile, "MNT")
    ENDIF

  CASE m.cType = "R"
    IF EMPTY(This.cReportDir)
      This.cReportDir = GETDIR("",
        "Specify report folder for + ;
        "project","Report folder")
    ENDIF

    IF NOT EMPTY(This.cReportDir)
      cDestFolder = This.cReportDir
      cPartnerFile = FORCEEXT(m.cFile, "FRT")
    ENDIF

  CASE m.cType = "X" AND ;
      INLIST(UPPER(JUSTEXT(m.cFile)), ;
      "ANI", "BMP", "CUR", "DIB", "EMF", ;
      "EXIF", "GIF", "GFA", "ICO", "JPG", ;
      "JPEG", "JPE", "JFIF", "PNG", "TIF", ;
      "TIFF", "WMF")
    IF EMPTY(This.cGraphicsDir)
      This.cGraphicsDir = GETDIR("",
        "Specify graphics folder for + ;
        "project","Graphics folder")
    ENDIF

    IF NOT EMPTY(This.cGraphicsDir)
      cDestFolder = This.cReportDir
      cPartnerFile = ""
    ENDIF

  CASE m.cType = "X" AND ;
      INLIST(UPPER(JUSTEXT(m.cFile)), "H")
    IF EMPTY(This.cIncludeDir)
      This.cIncludeDir = GETDIR("", ;
        "Specify include file folder for + ;
        " project","Include file folder")
    ENDIF

    IF NOT EMPTY(This.cIncludeDir)
      cDestFolder = This.cIncludeDir
      cPartnerFile = ""
    ENDIF
  ENDCASE

  IF EMPTY(m.cDestFolder)
    nSkipped = m.nSkipped + 1
  ELSE
    cDescription = oFile.Description

    cNewFile = FORCEPATH(m.cFile, ;
      m.cDestFolder)
    RENAME (m.cFile) TO (m.cNewFile)
    IF NOT EMPTY(m.cPartnerFile) AND ;
      FILE(m.cPartnerFile)
      RENAME (m.cPartnerFile) TO ;
        FORCEPATH(m.cPartnerFile, ;
```

```
        m.cDestFolder)
    ENDIF
    oFile.Remove()
    oFile = ;
      This.oProject.Files.Add(m.cNewFile)
    oFile.Description = m.cDescription
    REPLACE lMoved WITH .T. IN csrFilesInRoot

  ENDIF

ENDSCAN

RETURN "csrFilesInRoot"
```

The second tricky part is actually much harder. Files in a VFP project may contain references to other files in the project. Moving a file can break such references. For now, I've chosen to handle only the most common of those problems automatically. The ClassLoc field of form (SCX) and class library (VCX) files contains relative paths to the referenced objects (if they're on the same drive as the file itself); MoveFilesFromRoot calls another method, FixClassLoc (Listing 8), to correct those references. (The GetNewPath method it calls finds a new relative path to the location.) I also fix references to graphic files in Picture and Icon properties; the FixPictures method (not shown here) handles that chore.

Listing 8. VFP's forms and class libraries contain relative references to other class libraries. FixClassLoc fixes those references when the form or classlib is moved.

```
PROCEDURE FixClassLoc( ;
  cFileName, cOldDir, cNewDir)

LOCAL cCurPath, cNewPath, cFilePath
LOCAL nOldSelect

nOldSelect = SELECT()

cOldDir = ADDBS(m.cOldDir)
cNewDir = ADDBS(m.cNewDir)

cFilePath = ADDBS(JUSTPATH(m.cFileName))

SELECT 0
USE (m.cFileName) ALIAS __CXFile

SCAN
  IF NOT EMPTY(__CXFile.ClassLoc)
    cNewPath = This.GetNewPath( ;
      __CXFile.ClassLoc, m.cFilePath, ;
      m.cNewDir)
    REPLACE ClassLoc WITH m.cNewPath
  ENDIF
ENDSCAN

USE IN SELECT("__CXFile")

SELECT (m.nOldSelect)
RETURN
```

Other cases you need to watch for are coded references to other files that include relative paths, and include files. The method moves include files, but does not check other files in the project to see whether they reference the moved file.

Other listings

The class contains several other routines to produce lists. One builds a list of folders containing project files, including the number of files in each folder.

Another builds a list of files in the project that are stored outside the project's folder hierarchy; I wrote that one when I noticed a couple of folders from VFP's FFC hierarchy in the list of folders. Given that this project is stored on a network drive, my client was surprised to see anything that referenced the local hard drive.

A third method builds a list of files listed in the project that don't exist on disk, and a fourth builds a list of all the graphics referenced by Picture and Icon properties in the project. The code for these methods is straightforward, so I'm not including it here. (The complete class is included in this month's downloads.)

Copying a Project and its Files

When I told my client that there were thousands of extraneous files in the project folders, he suggested that the easiest way to clean up would be to copy the project and the files it's using into a new folder, rather than deleting all the junk. So I added a method (CopyProject, shown in Listing 9) to the project to do just that. The only tricky part is ensuring that the memo files come along with the various table-based files. The version of this method in the downloads brings along additional properties of the project, ensures that graphics files referenced in the project are copied, whether or not they're included in the original project, and makes sure the right file is set as the main program.

Listing 9. The CopyProject method makes a copy of the project and all its files in the specified folder.

```
PROCEDURE CopyProject(cNewRoot)

LOCAL oFile, cNewName, cNewPath, cNewProject
LOCAL oNewProject, cMissing, cNoAdd
LOCAL 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.
cNewProject = ;
  FORCEPATH(This.oProject.Name, m.cNewRoot)
CREATE PROJECT (m.cNewProject) NOWAIT
oNewProject = _VFP.ActiveProject

* Copy all files to appropriate directories
cMissing = ""
cNoAdd = ""
```

```
FOR EACH oFile IN This.oProject.Files
  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 = 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)
  CATCH
    cNoAdd = m.cNoAdd + ;
      CHR(13) + CHR(10) + m.cNewName
  ENDTRY

ENDFOR

* Report to user
IF NOT EMPTY(m.cMissing)
  cMissing = "The following files in the " + ;
    "project were not found: " + m.cMissing
ENDIF

IF NOT EMPTY(m.cNoAdd)
  cNoAdd = "The following files were not " + ;
    "able  to be added to the project: " + ;
    m.cNoAdd
ENDIF

DO CASE
CASE EMPTY(m.cMissing) AND EMPTY(m.cNoAdd)
  cMessage = "Project and all its files " + ;
    "copied to " + m.cNewRoot
CASE EMPTY(m.cMissing)
  cMessage = "All project files were " + ;
    "copied. " + m.cNoAdd
CASE EMPTY(m.cNoAdd)
  cMessage = m.cMissing
OTHERWISE
  * We have both kinds of message.
  cMessage = m.cMissing + ;
    CHR(13) + CHR(10) + m.cNoAdd
ENDCASE

MESSAGEBOX(m.cMessage)

* Close the new project
oNewProject.Close()

RETURN
```

The method first creates the project file (creating the folder, if necessary). Then it copies each file into the new hierarchy, again creating folders as needed. After copying the file, the new file is added to the new project.

Along the way, the method keeps track of files it can't find, and files that couldn't be added to the project. After the main loop, these two lists are displayed. Finally, the new project is closed.

Producing Reports

While I like putting results into cursors, after I'd written a couple of the list routines, it became apparent that I'd also want a way to put the results into reports. But I didn't want to have to design a separate report for each list.

I decided to take advantage of VFP's Quick Report capability, and allow the reporting methods to generate a report if the necessary report didn't exist. So I wrote a generic reporting method (ReportOnProject, shown in Listing 10) that accepts three parameters: an alias containing the data to report, a comma-separated list of the fields in that alias to report on, and the name of a report file. The last two parameters are optional.

Listing 10. The ReportOnProject method provides a generic report mechanism for the tool. It can use an existing FRX or create one on the fly.

```
PROCEDURE ReportOnProject( ;
  cAlias, cColumns, cReport)

IF VARTYPE(m.cAlias) <> "C" OR ;
   NOT USED(m.cAlias)
  ERROR 11
ENDIF

LOCAL cReportFile, lCreatedReport
LOCAL cFieldsClause, aFieldList[1]
LOCAL nFields, nField, cFieldList
LOCAL aColumns[1], nColumns
LOCAL cFieldName, cNewFieldSpec

IF VARTYPE(m.cReport) <> "C" OR ;
   NOT FILE(m.cReport)
  cReportFile = ;
    FORCEPATH("ProjectReport",SYS(2023))
  lCreatedReport = .T.

  IF NOT EMPTY(m.cColumns)
    nColumns = ALINES(aColumns, ;
      m.cColumns, ",")
  ELSE
    nColumns = 0
  ENDIF

  * Handle memo fields, so that they get
  * created wide enough
  nFields = AFIELDS(aFieldList, m.cAlias)
```

```
  cFieldList = ""

  FOR nField = 1 TO nFields
    cFieldName = aFieldList[m.nField, 1]
    IF m.nColumns = 0 OR ASCAN(aColumns, ;
       m.cFieldName, -1, -1, -1, 7) <> 0
      * It's in the list of columns to include
      IF aFieldList[m.nField,2] = "M"
        * It's a memo field, so convert it to
        * a long char for generating the
        * report.
        cNewFieldSpec = "CAST(" + ;
          m.cFieldName + " AS C(250)) AS " + ;
          m.cFieldName
      ELSE
        * Just pass it along as is.
        cNewFieldSpec = m.cFieldName
      ENDIF
      cFieldList = m.cFieldList + "," + ;
        m.cNewFieldSpec
    ENDIF
  ENDFOR

  cFieldList = SUBSTR(m.cFieldList, 2)

  * Now run the query
  SELECT &cFieldList ;
    FROM (m.cAlias) ;
    INTO CURSOR csrForQuickReport

  IF VARTYPE(m.cColumns) = "C" AND ;
     NOT EMPTY(m.cColumns)
    cFieldsClause = "FIELDS " + m.cColumns
  ELSE
    cFieldsClause = ""
  ENDIF

  CREATE REPORT (m.cReportFile) ;
    FROM csrForQuickReport ;
    COLUMN &cFieldsClause.

  * Kill the special cursor
  USE IN SELECT("csrForQuickReport")

ELSE
  cReportFile = m.cReport
  lCreatedReport = .F.
ENDIF

SELECT (m.cAlias)
REPORT FORM (m.cReportFile) PREVIEW

* Clean up
IF m.lCreatedReport
  ERASE (FORCEEXT(m.cReportFile, "FRX"))
  ERASE (FORCEEXT(m.cReportFile, "FRT"))
ENDIF

RETURN
```

The biggest challenge here was handling memo fields. I use them for filenames and paths in many of the results cursors because there's no way to know how long those could be. But VFP's quick report allocates just a very narrow space for memo fields; while it properly sets the stretch and float attributes, the column was nowhere near as wide as I wanted it to be. To work around this, ReportOnProject creates a new cursor, replacing all memo fields with long character fields, and uses that as the basis for the quick report. However, the report actually runs against the original cursor.

To produce reports, I have a method corresponding to each of the List methods. Each of these Report methods receives the alias of the cursor to report on and, optionally, the name of the report file. Each method runs a query against the specified cursor to pull out the data to report and put it in the desired order. Then each method calls ReportOnProject. For example, ReportFilesInRoot, shown in Listing 11, reports on the results of the ListFilesInRoot method. Note that the reporting methods assume that you've run the corresponding List method first.

Listing 11. Each of the List methods has a corresponding Report method to produce a formatted list.

```
PROCEDURE ReportFilesInRoot(cAlias, cReport)

SELECT mFileName, ;
       LEFT(mFileName, 250) as cFileName ;
       FROM (m.cAlias) ;
       ORDER BY 2 ;
       INTO CURSOR csrReport

This.ReportOnProject("csrReport", ;
  "mFileName", m.cReport)

USE IN SELECT("csrReport")

RETURN
```

Because you can't sort a query on a memo field, I extract a long stretch of the filename and sort on that. But the second parameter to ReportOnProject ensures that only the memo field shows up in the report.

What's Next?

The obvious thing that's missing from this set of tools is a user interface. It wouldn't be hard to build a form that lets you select the project you want to work with and click buttons to run the various methods. The form could provide a view into the project as well.

I suspect I'll continue to add methods as needs arise. Let me know what methods you add, or what problems you'd like to solve with this 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.comc