May, 2007

Advisor Answers

Clean up a Project

Visual FoxPro 9/8/7

Q: I've just inherited a VFP application for maintenance. Both the project itself and the project directories seem to contain a lot of files that aren't actually used in the application. How can I clean up so I know what I'm working with?

A: This is a common situation with projects that have been around for a while. Code gets replaced, test programs are written and left around, and so forth. Fortunately, it's fairly easy to clean up in most cases.

The first step is to create a new project (CREATE PROJECT NewApp) and add the main program for the application to it. Then click on the Build button and choose Rebuild project. That pulls into the project all the files used by the main program and the programs it calls (recursively to the bottom of the chain). There is one caveat, which is that files referenced only indirectly (by macro expansion or name substitution) aren't pulled into the project. So you have to test carefully to make sure you haven't missed any files.

You can use some code to help figure out whether you've omitted any files of importance. The following code builds a cursor of all the files found in one project that aren't found in another. Pass the project names, including paths, as parameters.

```
LPARAMETERS cOldProject, cNewProject
LOCAL oOld as VisualFoxpro.IFoxProject, ;
      oNew as VisualFoxpro.IFoxProject
MODIFY PROJECT (m.cOldProject) NOWAIT
oOld = _VFP.ActiveProject
MODIFY PROJECT (m.cNewProject) NOWAIT
oNew = _VFP.ActiveProject
CREATE CURSOR Missing (mFile M)
LOCAL oFile, oNewFile, cFileName
FOR EACH oFile IN oOld.Files
   * Look for each file from the old project
   * in the new project. The filename without path
   * is the key in the collection.
   cFileName = JUSTFNAME(oFile.Name)

   TRY
      oNewFile = oNew.Files[m.cFileName]
```

```
   CATCH
      * Used in old, not in new
      INSERT INTO Missing VALUES (oFile.Name)

   ENDTRY
ENDFOR
```

This code uses the Project object and its Files collection. Whenever a VFP project is opened, a Project object is created. You can address the active project using _VFP.ActiveProject. To get the whole set of open projects, use the collection _VFP.Projects.

The Project object has a Files collection containing all the files in the project. Each file in the collection has a key which is the file name without path (stem plus extension). The code here grabs a file from one project and tries to access the file with the same key in the other project. The use of TRY-CATCH eliminates a large brute force loop.

When this program is done, the cursor Missing contains a list of all files in the old project that aren't in the new one.

Once you've created a new project, the next step is to move the project and its code into a new directory, keeping only the code that's actually used. There are two ways to approach this task; both address the project as an object.

The first approach is to run code to copy all the files referenced in a project into a new folder structure. The following code accepts the name (including path) of the project, the original path and the new path. It then opens the project and reads the list of files. For each, if a same-named file doesn't already exist in the new folder hierarchy, the file is copied. Folders are created as needed. For those VFP components that involve two files (forms, class libraries, and menus here), the memo file is also copied. (This code doesn't address memo files for reports and labels because the project for which it was written didn't include any.) Once this code has finished, you can copy the new project into the new folder and rebuild it.

```
LPARAMETERS cProject, cOriginalPath, cNewPath
LOCAL oProject, oFile, cNewName, cNewFilePath
MODIFY PROJECT (cProject) nowait
oProject = _vfp.ActiveProject
* Copy all files to appropriate directories
FOR EACH oFile IN oProject.Files
   cNewName = cNewPath + STREXTRACT(oFile.Name, ;
              cOriginalPath,"",1,3)
   cNewFilePath = JUSTPATH(cNewName)
   IF NOT FILE(cNewName)
```

```
      IF NOT DIRECTORY(cNewFilePath)
         MD (cNewFilePath)
      ENDIF
      COPY FILE (oFile.Name) TO (cNewName)
   ENDIF
   IF JUSTEXT(cNewName) = "scx" AND ;
      NOT FILE(FORCEEXT(cNewName, "SCT"))
      COPY FILE (FORCEEXT(oFile.Name, "SCT")) TO  ;
                (FORCEEXT(cNewName, "SCT"))
   ENDIF
   IF JUSTEXT(cNewName) = "vcx" AND ;
      NOT FILE(FORCEEXT(cNewName, "VCT"))
      COPY FILE (FORCEEXT(oFile.Name, "VCT")) TO  ;
                (FORCEEXT(cNewName, "VCT"))
   ENDIF
   IF JUSTEXT(cNewName) = "mnx" AND ;
      NOT FILE(FORCEEXT(cNewName, "MNT"))
      COPY FILE (FORCEEXT(oFile.Name, "MNT")) TO  ;
                (FORCEEXT(cNewName, "MNT"))
   ENDIF
ENDFOR
```

The second approach is to copy everything to the new folders and then eliminate those that are not used in the project. The following code builds a list of suspect files. It accepts the name, including path of a project, and a list of folders to check. It first builds an array listing all the files in the project. Then, it goes through all the directories listed in the second parameter; in each, it checks every file against the list of files in the project. If the file isn't in the project, it adds it to a cursor. When this code is done, you can check the cursor and decide which files to delete.

```
LPARAMETERS cProject, cPath
* Look for unused code
LOCAL oProject, nCounter, oFile
LOCAL nDirs, nDir, aDirs[1]
MODIFY PROJECT (m.cProject) nowait
oProject = _VFP.ActiveProject
* First, make a list of all files in project
LOCAL aProjFiles[oProject.Files.Count]
nCounter = 0
FOR EACH oFile IN oProject.Files
   nCounter = m.nCounter + 1
   aProjFiles[m.nCounter] = UPPER(oFile.Name)
ENDFOR
CREATE CURSOR Unused (mFileName M)
* Now traverse directories
* First, make a list of directories
nDirs = ALINES(aDirs, m.cPath, 1, ";", ",")
CREATE CURSOR DirsToCheck (mDirName M)
FOR nDir = 1 TO m.nDirs
   INSERT INTO DirsToCheck VALUES (aDirs[m.nDir])
ENDFOR
```

```
LOCAL aFiles[1], cOldDir, cFile, nFilesToCheck, cExt
cOldDir = SET("Default") + CURDIR()
SCAN
   IF DIRECTORY(mDirName)
      CD ALLTRIM(mDirName)
      nFilesToCheck = ADIR(aFiles, "*.*")
      FOR nFile = 1 TO m.nFilesToCheck
         cFile = aFiles[m.nFile, 1]
         cExt = JUSTEXT(m.cFile)
         IF INLIST(cExt, "PRG", "SCX", "MNX", ;
                   "FRX", "VCX", "QPR")
            IF ASCAN(aProjFiles, FORCEPATH(m.cFile, ;
                ALLTRIM(mDirName)), -1, -1, 1, 7) = 0
               INSERT INTO Unused ;
                 VALUES (FORCEPATH(m.cFile, ;
                         DirsToCheck.mDirName))
            ENDIF
         ENDIF
      ENDFOR
   ENDIF
ENDSCAN
CD (m.cOldDir)
RETURN
```

This month's Professional Resource CD contain all three programs above, as ListMissingFiles.PRG, CopyProject.PRG, and CheckForUnusedCode.PRG, respectively.

Cleaning away years of accumulation can help a lot when you start working with an existing project. Having a new folder structure and a project containing only code that's actually used makes it easier for you to figure out how the code works and what needs to be done.

–Tamar

Find Out Where a Class is Used

VFP 9/8

Q: I want to find all the places in a project where I've used a particular class. Is there an easy way to do this?

–Dmitry Litvak (via the Internet)

A: My first instinct was to send you to Code References. This tool, introduced in VFP 8, searches a project or folder structure for a specified string and shows you all the places it's found. I use it extensively, especially when working with other people's code.

However, it turns out that Code References doesn't check the class of items it encounters in a form or class. It looks at the properties and methods and at the name of the object, but not the class or class library.

Because the source code for Code References is provided with VFP (in the Tools\XSource\VFPTools\FoxRef folder-unzip Tools\XSource\XSource.Zip if you haven't already done so), you can modify the tool to check the class and/or class library. This change turns out to be remarkably easy. To search the class name, just add the following code to the DoSearch method of RefSearchForm in FoxRefSearch_Form.PRG:

```
IF !EMPTY(Class)
m.lSuccess = THIS.FindInText(Class, FINDTYPE_NAME, ;
   NVL(cRootClass, cClassName), cObjName, ;
   SEARCHTYPE_EXPR, UniqueID, "CLASS", .T.)
ENDIF
```

The code goes after the block it's based on, which checks the object name (ObjName). Be aware that this code doesn't let you change the class name using Code References, but that's a good thing, since such changes have major consequences. (To change the name of a class, use the Class Browser and make sure everything that references that class is open in the Browser at the same time.) To search the class library, as well, add another analogous block that passes the ClassLoc field to FindInText.

To use the updated tool, rebuild the Code References application and set _FoxRef to point to your new APP file. An updated version of FoxRefSearch_Form.PRG that searches in the Class field is included on this month's Professional Resource CD.

If changing the tool makes you uncomfortable, you can write your own code to do the search instead. You can use the project object and its Files collection, like this:

```
LPARAMETERS cProject, cSearchClass
LOCAL oFile, cUpperClass
MODIFY PROJECT (cProject) NOWAIT
cUpperClass = UPPER(m.cSearchClass)
CREATE CURSOR Matches ;
   (mFileName M, nRecord N, mObjName M)
FOR EACH oFile IN _VFP.ActiveProject.Files
   * If it's a form or classlib
   IF INLIST(oFile.Type, "K", "V")
      * Open the file and search
      SELECT 0
```

```
      USE (oFile.Name) ALIAS __CXFile

      SCAN FOR UPPER(Class) == m.cUpperClass
         * If inside loop, found a match. Save it.
         INSERT INTO Matches ;
            VALUES (oFile.Name, RECNO("__CXFile"), ;
                    __CXFile.ObjName)
      ENDSCAN

      USE IN SELECT("__CXFile")
   ENDIF
ENDFOR
RETURN
```

When this code finishes, the cursor Matches contains a list of objects in the project that use the specified class. Each record in the cursor indicates the file, the object name and the record number in the file.

The code uses a mixed approach, treating the project as an object, but treating the individual forms and class libraries as tables. It opens the project and traverses the Files collection. When it encounters a form (type "K") or a class library (type "V"), it opens the SCX or VCX as a table and loops through the records. This program is included on this month's PRD as SearchForClass.PRG.

The two approaches to this problem highlight two of VFP's strengths. Having source code for so many of the tools that come with the product means we can tweak them when we need to. The open architecture that lets us write code to process projects and their files means we can supplement the provided tools with our own.

–Tamar