September, 2000

Use Visual FoxPro to Change and Audit Word Documents

Imagine that you need to manage a large number of Microsoft Word documents, all of which require various changes. Some of the changes require search-and-replace, while others involve overall formatting, such as changing the margins. Perhaps you also want to log information about the documents such as the list of graphic items they contain.

How would you approach such a task? If only a few documents were involved, performing these chores manually would be no big deal. But once you get past five or six documents, ensuring that every document gets every necessary change starts to get difficult.

You could create macros in Word to simplify things. That would probably keep the task manageable if you're working with no more than 20 or 30 documents. But when I faced this task, I had approximately 800 documents to deal with. I was in the final stages of creating the Hacker's Guide to Visual FoxPro 6.0 (with Ted Roche, Hentzenwerke Publishing, 1998) and we had a separate document for each entry in the reference section of the book. We needed to do all the chores mentioned in the first paragraph, as well as many others. (See the Editor's View column in the January '99 issue of FoxPro Advisor for some of the specific details; you can read it online at www.advisor.com.)

Clearly, no manual approach was going to work. Instead, I used the tool I knew best, Visual FoxPro, and let it do all the heavy lifting. In this article, I'll show you a class that lets you audit and make changes to multiple documents, with no wear and tear on your carpal tunnel.

The plan

The strategy is to use a class to handle three discrete tasks. First, the class manages an automated instance of Microsoft Word, ensuring that it exists and making it available when needed. Second, it keeps track of the group of documents on which we want to operate. Finally, it offers a set of operations to perform on those documents. We'll look at each of those tasks separately.

The class is called cusProcessDocuments and is subclassed from the _custom class in the FoxPro Foundation classes. You'll find the

complete code for this class on this month's Professional Resource CD and at www.advisor.com.

Managing Word

The class has a custom property, oWord, to hold a reference to a Word Automation server. Three custom methods manage the server: GetWord, ReleaseWord and CheckWordAndDoc.

GetWord loads the Word Automation server and stores a reference to it in oWord. GetWord uses the Registry class that comes with VFP 6 to make sure that it can find the Word automation object.

Here's the code for GetWord:

```
* Create an instance of the Word Automation server
* and store a reference to it in This.oWord
LOCAL oRegistry
oRegistry = NewObject("Registry",HOME() + "FFC\Registry")
IF oRegistry.IsKey("Word.Application")
   This.oWord = CreateObject("Word.Application")
   IF VarType(This.oWord)<>"O"
      This.TellUser("Can't create Word automation object")
   ENDIF
ENDIF
RETURN
```

The ReleaseWord method closes the Word server and sets oWord to .null. The Destroy method calls ReleaseWord to ensure that the server gets shut down, if oWord holds a reference to the server at that point. (It might not, since the class can be used to manipulate files without opening Word.)

The CheckWordAndDoc method determines whether oWord holds a reference to an active Word server and whether the server has an active document. It returns a logical value. Processing methods can call CheckWordAndDoc to determine whether there's anything to process in order to avoid errors.

Tracking documents

The class is designed to work on a group of files. A number of custom properties determine the list:

- aFileList–an array property to hold the list of files currently being processed.
- cDirectory–the directory, including full path, from which aFileList should be filled.

- cCurrentListDir–the directory, including full path, from which aFileList was last filled. (Protected)
- cFileMask–the file mask for the files to be loaded. Defaults to "*.*".
- cOmitExtensions–a comma-delimited list of extensions that should be excluded from the list of files. Defaults to "gif,bmp".
- nSortOrder–the order to sort the list, based on the columns of ADIR().

A number of methods manage the list of files:

- GetFiles–loads aFileList based on cDirectory and cOmitExtensions.
- IsValidDir–returns a logical value that indicates whether a directory passed as a parameter is valid. Used by GetFiles to check the validity of cDirectory.
- AnyFiles–returns a logical value that indicates whether there are currently any files in aFileList.
- PutFilesInCursor–moves the list of files from aFileList into a cursor called AllFileInfo. Returns the number of records in the cursor. Used by many methods to simplify processing.

The GetFiles method fills aFileList with information about all files in cDirectory that match cFileMask, except those with the extensions specified in cOmitExtensions. (I'm in the habit of using extensions other than .DOC for my Word documents to indicate different versions of the same file, so didn't want to be limited to looking for .DOC files.) The method also sets cCurrentListDir to the same value as cDirectory. Doing so ensures that we can find the original directory later on, even if cDirectory has changed. GetFiles returns the number of files loaded. Here's the code for GetFiles:

```
* Fill aFileList with the list of files to be processed
* based on the cFileMask property. Sort them based
* on the nSortOrder property.
LOCAL nFileCount, cFullMask, nValidDir, nCount, nDelCount
IF EMPTY(This.cDirectory)
  cFullMask = This.cFileMask
ELSE
  * Check that specified directory is valid.
  nValidDir = This.IsValidDir(This.cDirectory, .T.)
  IF nValidDir <> 0
    This.TellUser("Specified directory doesn't exist")
    RETURN 0
  ENDIF
  cFullMask = ADDBS(This.cDirectory)+This.cFileMask
ENDIF
nFileCount = ADIR(This.aFileList, cFullMask)
```

```
This.cCurrentListDir = This.cDirectory
IF nFileCount = 0
  DIMENSION This.aFileList[1]
  This.aFileList[1] = ""
ELSE
  * Get rid of any that have the wrong extension
  IF NOT EMPTY(This.cOmitExtensions)
    nDelCount = 0
    FOR nCount = nFileCount TO 1 STEP -1
      IF UPPER(JustExt(This.aFileList[ ;
               nCount, ADIR_NAME])) $ ;
        UPPER(This.cOmitExtensions)
        * Delete it
        ADEL(This.aFileList, nCount)
        nDelCount = nDelCount + 1
      ENDIF
    ENDFOR
    nFileCount = nFileCount - nDelCount
    DIMENSION This.aFileList[ nFileCount, ;
                ALEN(This.aFileList,2) ]
  ENDIF

  * Now sort them
  ASORT(This.AFileList, This.nSortOrder, nFileCount)
ENDIF
RETURN nFileCount
```

Processing Documents

The methods used to process documents can be broken down into three categories: utility methods to interact with Word, utility methods to interact with the processing class, and methods that actually accomplish something.

The first group, Word utility methods, includes OpenDocument and CloseDocument. As their names suggest, these methods are wrappers for the Word Document object's Open and Close methods, respectively. In each case, the name of the document is passed as a parameter, and the wrapper method provides the other parameters to pass to the corresponding Word method. Since the concept of these methods is easy to understand, see the source code for the implementation of the wrapper methods.

The second group, utility methods for the processing class, is really at the heart of this article. We'll come back to these in a bit.

The third group, methods that do something to the documents, are the ones that you'll provide. We'll look at a couple of examples here, but the basic idea is that you write a method for each task you want to perform.

Some tasks involve finding all occurrences of some type of item (such as graphics or tables) in a document. Methods for these tasks generally create a log of what they find. Each log is created as a cursor and is left open after the method finishes. The class includes a couple of methods for processing those log files. (See the "Final Thoughts" section for some notes on these methods.)

Turning off revision tracking

Here's a very simple method for an often-used task. RevisionsOff turns off revision tracking. It accepts a logical parameter that determines whether revisions should be accepted before turning tracking off.

```
* Turn off revision marks.
* Accepts a parameter to indicate
* whether to accept revisions first.
LPARAMETER lAccept
  * lAccept = Accept revisions before turning off?
* First, make sure we have Word and a document
IF NOT This.CheckWordAndDoc()
  RETURN .F.
ENDIF
* Now go for it.
IF lAccept
  This.oWord.ActiveDocument.AcceptAllRevisions()
ENDIF
This.oWord.ActiveDocument.TrackRevisions = .F.
RETURN
```

Note that the method neither opens nor closes the document. This is essential for performing the kind of multi-task, multi-document processing we want to do.

Search and Replace

One of my major motivations for writing the processing class was to do bulk search-and-replace operations. The ReplaceText method does a series of searches in a single document. It uses a table that indicates what to look for and what to replace it with. Table 1 shows the structure of the table.

Table 1. Defining search and replace – The fields of this table provide the information for a series of search and replace operations to be performed on one or more documents.

Original	Character	20	The search string
Replment	Character	20	The replacement string
lReplace	Logical	1	Replace it (vs. log it only)
lCase	Logical	1	Is the search case-sensitive?
lFormattin	Logical	1	Consider formatting in the search?
cFont	Character	20	Search for what font?
lBold	Logical	1	Search for bold text?
lItalic	Logical	1	Search for italic text?
nSize	Numeric	2	Font size to search for
cStyle	Character	30	Style to search for
nColor	Numeric	3	Text color to search for
cDescrip	Character	25	Description of this search item

One of Word's strengths is that you can search for text, formatting or a combination of the two. The ReplaceText method lets you specify a

search string, and indicate whether or not the search is case-sensitive and whether or not formatting counts. If you choose to include formatting, you can indicate the font and size, a few font characteristics (bold, italic and color) and the style of the text. To search for formatting only, set Original to the empty string, set lFormattin to .T. and set the appropriate formatting fields. The last field (cDescrip) is for your purposes only, to help you understand the table contents.

The ReplaceText method accepts three parameters. The first, lDoIt, is logical and indicates whether to actually perform the replacements or just log the found strings. Pass .F. to perform a search-and-log only. The second parameter, cLogAlias, specifies an alias for the log cursor to be created. If it's omitted, the cursor's alias is named "Replaced." The third parameter, cReplaceTable, indicates the name of the table containing the search information. If it's omitted, the table named in the custom cReplaceTable property is used.

The requirement to keep a log of the changes makes the method more complicated than it would be otherwise. Without a log, you could set up Word's Find object and call its Execute method once. Keeping track of the changes means that you have to find each occurrence, log the original string, then perform the replacement, and log the changed string. Here's the code for ReplaceText:

```
* This method processes the open document in the
* Word Automation object by going through a table
* and replacing each string in one field of the table
* with the specified string in another field.
LPARAMETER lDoIt, cLogAlias, cReplaceTable
  * lDoIt = Indicates whether to actually perform
  *      replacement or just log matches.
  *      .T. = perform replacement.
  *      (But data in replacement table
  *      overrides this)
  * cLogAlias = name of log file. If omitted,
  *      use "Replaced"
  * cReplaceTable = name of table containing
  *      replacement information. If omitted,
  *      uses value from This.cReplaceTable.
  *      If a table is specified here, it must
  *      have the appropriate fields.
* Check alias parameter
IF VarType(cLogAlias)<>"C" AND NOT EMPTY(cLogAlias)
  This.TellUser( ;
    "Log file alias must be character or omitted")
ENDIF
IF NOT This.CheckWordAndDoc()
  RETURN .F.
```

```
ENDIF
* Now check replacements table
IF VarType(cReplaceTable) <> "C"
  cReplaceTable = This.cReplaceTable
ENDIF
IF USED("ReplMents")
  * Alias for replacement table is already in use
  * Close it without warning.
  USE IN ReplMents
ENDIF
IF NOT EMPTY(cReplaceTable)
  SELECT 0
  USE (cReplaceTable) ALIAS Replments
ENDIF
IF NOT USED("Replments")
  This.TellUser("Can't find replacements table")
  RETURN .F.
ENDIF
* If we get this far, we have a reference to Word
* with an open document, and an open replacements table.
* So we can start processing it.
* Get alias name
LOCAL cAlias
IF VarType(cLogAlias) = "C" AND NOT EMPTY(cLogAlias)
  cAlias = cLogAlias
ELSE
  cAlias = "Replaced"
ENDIF
IF NOT USED(cAlias)
  CREATE CURSOR (cAlias) ;
    (cDocument C(100), cOrigField C(20), ;
     cNewField C(20), cStyle C(30), lFormatting L, ;
     lCase L, cFont C(20), nSize n(3), lBold L, ;
     lItalic L, nColor N(3), mOldText M, mNewText M, ;
     tWhen T)
  SELECT Replments
ENDIF
LOCAL cFindData, cReplData, lFoundIt, lFoundAny, oRange
LOCAL cSentence, lStyleFound, nRangeStart, nRangeEnd
SCAN
  cFindData = TRIM(Original)
  cReplData = TRIM(Replment)

  * Search the document from top to bottom for the data item.
  oRange = This.oWord.ActiveDocument.Range()
  WITH oRange.Find
    * Remove formatting settings from previous
    * find/replace operation.
    .ClearFormatting()
    .Replacement.ClearFormatting()
    .Text = cFindData
    IF lFormattin   && formatting counts
      .Format = .T.
      WITH .Font
        * Only set if font if .T. in table;
```

```
      * otherwise doesn't matter
      IF lBold
        .Bold = .T.
      ENDIF
      IF lItalic
        .Italic = .T.
      ENDIF
      IF NOT EMPTY(nSize)
        .Size = nSize
      ENDIF
      IF NOT EMPTY(cFont)
        .Name = cFont
      ENDIF
      IF NOT EMPTY(nColor)
        .ColorIndex = nColor
      ENDIF
    ENDWITH
    IF NOT EMPTY(cStyle)
      * Need to make sure the specified style exists
      * in this document
      lStyleFound = .F.
      FOR EACH oStyle IN This.oWord.ActiveDocument.Styles
        IF UPPER(oStyle.NameLocal) = ;
           UPPER(ALLTRIM(cStyle))
          lStyleFound = .T.
          .Style = cStyle
          EXIT
        ENDIF
      ENDFOR
      IF NOT lStyleFound
        * Can't possibly find any matches, so get out
        * of this loop pass
        LOOP
      ENDIF
    ENDIF
  ELSE
    .Format = .F.
  ENDIF
  .Replacement.Text = ""
  .Forward = .T.
  .Wrap = 0  && wdFindStop
  IF lCase
    .MatchCase = .T.
  ELSE
    .MatchCase = .F.
  ENDIF
  .MatchWholeWord = .F.
  .MatchWildcards = .F.
  .MatchSoundsLike = .F.
  .MatchAllWordForms = .F.
ENDWITH

WITH This.oWord
  * Find all matches and log them.
  * Because we want to log the original and changed
```

```
    * strings, need to find them one at a time.
    lFoundIt = oRange.Find.Execute()
    lFoundAny = lFoundIt
    DO WHILE lFoundIt
      * Grab the containing sentence
      nRangeStart = oRange.Start
      nRangeEnd = oRange.End
      oRange.Expand(wdSentence)
      cSentence = oRange.Text
      oRange.SetRange( nRangeStart, nRangeEnd )
      INSERT INTO (cAlias) ;
        VALUES (.ActiveDocument.FullName, ;
                cFindData, cReplData, ;
                Replments.cStyle, ;
                Replments.lFormattin, ;
                Replments.lCase, Replments.cFont, ;
                Replments.nSize, Replments.lBold, ;
                Replments.lItalic, Replments.nColor, ;
                cSentence, "", DATETIME())
      * Now replace it
      WITH oRange
        IF lDoIt and lReplace  && check global and local
          .Find.Replacement.Text = cReplData

          .Collapse(wdCollapseStart)
          lFoundIt = .Find.Execute(,,,,,,,,,, ;
                                   wdReplaceOne)

          * Log replacement
          nRangeStart = oRange.Start
          nRangeEnd = oRange.End
          oRange.Expand(wdSentence)
          cSentence = oRange.Text
          oRange.SetRange( nRangeStart, nRangeEnd )
          REPLACE mNewText WITH cSentence IN (cAlias)
        ENDIF
        .Collapse(wdCollapseEnd)
      ENDWITH

      * Are we done?
      IF oRange.End = ;
         This.oWord.ActiveDocument.Characters.Count - 1
        lFoundIt = .F.
      ELSE
        * If not, find next occurrence
        lFoundIt = oRange.Find.Execute()
      ENDIF
    ENDDO
  ENDWITH
ENDSCAN
* Close table of items to replace to allow this method
* to be called with different lists
USE IN Replments
RETURN
```

The set of fields in the search table doesn't use all the capabilities of Word's search engine. In particular, while it allows you to specify particular styles, colors and formatting to find, it doesn't allow you to change them. Word is capable of doing so. You might want to enhance the table to include both search and replace versions of those fields. (In fact, the method doesn't handle a number of other capabilities of Word's search engine, but they're less commonly used.)

Logging document contents

In addition to information content, documents have structure. Word's Automation model lets you explore this structure. For the Hacker's Guide, we needed to look at many different aspects of the underlying structure. One of them was the kind of graphic objects in the document.

The LogGraphics method creates a cursor listing graphics, indicating for each, whether it's linked, whether it's embedded and, for linked graphics, the name of the underlying file. LogGraphics accepts two parameters. The first is numeric and indicates what kind of graphics to log. The method can log all graphics, only linked, non-embedded graphics, or only embedded graphics. The second, optional, parameter is a comma-delimited list of files to log. If it's omitted, all graphic files are logged.

The method traverses the InlineShapes and Shapes collections, which track objects on the text layer and on the drawing layer (in front of the text layer), respectively, to find pictures. Here's the code:

```
* Create a log of graphic items.
* Indicate the graphic and whether it's linked,
* embedded or both. Accepts parameters indicating
* whether to log all or only one kind, and listing
* specific graphics to log.
* Returns number of graphics logged.
LPARAMETERS nLogWhat, cGraphics
  * nLogWhat - Which graphics should be logged.
  *  0 or omitted = log all graphics
  *  1 = log only linked and not embedded graphics
  *  2 = log only embedded graphics
  * cGraphics - Comma-delimited list of graphic files
  *             to log. If omitted, log all graphics.
  *             Note that only linked graphics can be
  *             checked by name.

LOCAL nLogCount, lIsEmbedded, lLogIt, lLogAll
* First make sure we have Word and a document available
IF NOT This.CheckWordAndDoc()
  RETURN .F.
```

```
ENDIF
* Check parameter value
IF VarType(nLogWhat) = "L"
  nLogWhat = 0
ELSE
  IF VarType(nLogWhat) <> "N"
    This.TellUser(;
      "Pass 0, 1 or 2 to indicate graphic type to log")
    RETURN .F.
  ENDIF
ENDIF
* Check graphic files list
IF VarType(cGraphics) = "C"
  lLogAll = .F.
ELSE
  lLogAll = .T.
ENDIF
* Create the log file
IF NOT USED("Graphics")
  CREATE CURSOR Graphics ;
    (cDocument C(100), cGraphic C(100), ;
     lLinked L, lEmbedded L, tWhen T)
ENDIF
* Now find the graphics.
nLogCount = 0
WITH This.oWord.ActiveDocument
  FOR EACH oShape IN .InLineShapes
    DO CASE
    CASE oShape.Type = wdInlineShapeLinkedPicture
      * Linked, may or may not be embedded
      * Figure out whether to log it.
      lIsEmbedded = ;
        oShape.LinkFormat.SavePictureWithDocument
      DO CASE
      CASE nLogWhat = 0
        lLogIt = .T.
      CASE nLogWhat = 1 AND NOT lIsEmbedded
        lLogIt = .T.
      CASE nLogWhat = 2 AND lIsEmbedded
        lLogIt = .T.
      OTHERWISE
        lLogIt = .F.
      ENDCASE
      IF NOT lLogAll
        lLogIt = ;
          UPPER(JustStem(oShape.LinkFormat.SourceName)) ;
          $ UPPER(cGraphics)
      ENDIF

      IF lLogIt
        nLogCount = nLogCount + 1
        INSERT INTO Graphics ;
          VALUES (.FullName, ;
                  oShape.LinkFormat.SourceFullName, ;
                  .T. , lIsEmbedded, DATETIME())
```

```
    ENDIF

  CASE oShape.Type = wdInlineShapePicture AND lLogAll
    * Don't bother if we're looking for specific files
    * Embedded!
    IF nLogWhat <> 1
      INSERT INTO Graphics ;
        VALUES (.FullName, "", .F. , .T., DATETIME())
      nLogCount = nLogCount + 1
    ENDIF

  OTHERWISE
    && Not interested in these
  ENDCASE
ENDFOR

FOR EACH oShape in .Shapes
  DO CASE
  CASE oShape.Type = msoLinkedPicture
    * Linked, may or may not be embedded
    lIsEmbedded = ;
      oShape.LinkFormat.SavePictureWithDocument
    DO CASE
    CASE nLogWhat = 0
      lLogIt = .T.
    CASE nLogWhat = 1 AND NOT lIsEmbedded
      lLogIt = .T.
    CASE nLogWhat = 2 AND lIsEmbedded
      lLogIt = .T.
    OTHERWISE
      lLogIt = .F.
    ENDCASE
    IF NOT lLogAll
      lLogIt = ;
        UPPER(JustStem(oShape.LinkFormat.SourceName)) ;
        $ UPPER(cGraphics)
    ENDIF

    IF lLogIt
      nLogCount = nLogCount + 1
      INSERT INTO Graphics ;
        VALUES (.FullName, ;
                oShape.LinkFormat.SourceFullName, ;
                .T. , lIsEmbedded, DATETIME())
    ENDIF

  CASE oShape.Type = msoPicture AND lLogAll
    * Embedded, so don't bother if we're
    * looking for specific files
    IF nLogWhat <> 1
      INSERT INTO Graphics ;
        VALUES (.FullName, "", .F. , .T., DATETIME())
      nLogCount = nLogCount + 1
    ENDIF
```

```
    OTHERWISE
      && Not interested in these
    ENDCASE
  ENDFOR
ENDWITH
RETURN nLogCount
```

Other methods might go through other collections or check for other characteristics of these collections.

Running multiple methods

To make the class really useful, we need to ability to apply multiple methods to multiple documents. That's the role of the ProcessFiles method. It runs a series of methods on all the documents in the aFileList array property.

ProcessFiles takes four parameters, as follows:

- aMethodsToApply is a two-column array listing the methods to be applied (such as ReplaceText, LogGraphics or other methods you write yourself). The name of each method to be applied to the documents goes in the first column and the parameters to pass go into the second as a comma-separated character string.
- lOpenFileFirst indicates whether the document should be opened before applying the list of methods. Some methods are meant to operate on files themselves rather than on document contents.
- lCloseFile indicates whether the document should be closed after applying all the methods.
- nSaveMode indicates whether the modified document should be saved after the methods have been run. There are three choices: don't save it, save it back to the same file or save it with a new filename. In the third case, the GetFileName method is called to create a new filename for the document. GetFileName returns a new name based on the old filestem (the name without the extension). A new extension is generated to indicate the file version. This is based on the Hacker's Guide and Advisor system of file naming, in which the first character of the extension indicates the version and is followed by the initials of the person creating the new version. The custom cUser property stores the user's initials, so the method strips off the extension, finds the first character and increments it, then puts the whole thing back together. You can modify this method to use your own naming convention.

ProcessFiles applies every method in aMethodsToApply to all the files listed in the aFileList property. It loops through the files, applying all the methods one at a time. Here's the code:

```
* This method traverses the list of files to be
* processed (from aFileList) and applies a set
* of methods to each. Opens Word and closes it
* when done, if necessary.
LPARAMETERS aMethodsToApply, lOpenFileFirst, ;
            lCloseFile, nSaveMode
  * aMethodsToApply = two-column array
  *   column 1 = the processing methods to call
  *              for each file. One method per row.
  *   column 2 = parameters to pass to the method. If
  *              lOpenFileFirst is .F., the parameters
  *              are passed after the file name.
  * lOpenFileFirst = should the file be opened before
  *                  the methods are called? Pass .T.
  *                  for things like ReplaceText and .F.
  *                  for methods that manipulate the
  *                  files themselves rather than their
  *                  contents.
  * lCloseFile = should the file be closed after the
  *              methods are done?
  * nSaveMode = should the file be saved after the
  *             methods and, if so, should it overwrite
  *             the original file or be saved as a new
  *             version?
  *             0 = don't save
  *             1 = save to same file
  *             2 = save to new file
LOCAL nCount, nFilesToProcess, lWordWasOpen,
LOCAL oActiveDocument, cExecuteString
LOCAL cFileName, cNewFileName, cParameters
* Check params
IF VarType(aMethodsToApply[1]) <> "C" OR ;
  EMPTY(aMethodsToApply[1])
  This.TellUser("No methods specified to apply")
  RETURN .F.
ENDIF
IF ALEN(aMethodsToApply,2) <> 2
  This.TellUser("List of methods must be two columns")
  RETURN .F.
ENDIF
IF VarType(lOpenFileFirst)<>"L"
  This.TellUser(;
    "lOpenFileFirst parameter must be logical")
  RETURN .F.
ENDIF
* Now check situation
IF NOT This.AnyFiles()
  This.TellUser(;
    "No files to process with specified methods")
  RETURN .F.
```

```
ENDIF
nFilesToProcess = ALEN(This.aFileList,1)
IF lOpenFileFirst AND VarType(This.oWord) = "O"
  lWordWasOpen = .T.
ELSE
  * Open Word and make a note of it
  This.GetWord()
  IF VarType(This.oWord) <> "O"
    This.TellUser(;
      "Can't open Word, so can't apply methods ")
    RETURN .F.
  ENDIF
  lWordWasOpen = .F.
ENDIF
FOR nCount = 1 TO nFilesToProcess
  cFileName = ADDBS(This.cDirectory) + ;
          This.aFileList[nCount,1]
  WAIT WINDOW "Processing " + cFileName NOWAIT
  IF lOpenFileFirst
    oActiveDocument = ;
      This.oWord.Documents.Open( cFileName )
  ENDIF

  * Apply each specified method to the open file
  FOR nMethod = 1 TO ALEN(aMethodsToApply, 1)
    * Construct command line
    cMethodToApply = aMethodsToApply[ nMethod, 1]
    cParameters = aMethodsToApply[ nMethod, 2]
    cExecuteString = cMethodToApply + "( "
    IF NOT lOpenFileFirst
      * add file name as first parameter
      cExecuteString = cExecuteString + cFileName
      IF NOT EMPTY(cParameters)
        * add comma
        cExecuteString = cExecuteString + ", "
      ENDIF
    ENDIF
    IF NOT EMPTY( cParameters )
      * add specified parameter list
      cExecuteString = cExecuteString + cParameters
    ENDIF
    * add trailing paren
    IF RIGHT(TRIM(cExecuteString),1) = "("
      * Macros don't like method name with empty paren,
      * so just use name
      cExecuteString = LEFT(cExecuteString, ;
                            LEN(TRIM(cExecuteString))-1)
    ELSE
      cExecuteString = cExecuteString + " )"
    ENDIF
    * Do it
    &cExecuteString
  ENDFOR

  DO CASE
```

```
  CASE nSaveMode = 0
    * do nothing

  CASE nSaveMode = 1
    * just save it
    oActiveDocument.Save()

  CASE nSaveMode = 2
    * get a new name and save to that file
    cNewFileName = This.GetFileName( cFileName )
    oActiveDocument.SaveAs( cNewFileName, ;
                            wdFormatDocument )
  ENDCASE

  IF lCloseFile
    oActiveDocument.Close( wdDoNotSaveChanges )
  ENDIF
ENDFOR
IF NOT lWordWasOpen
  This.ReleaseWord
ENDIF
RETURN .T.
```

You might choose to enhance this method by making the decisions about opening, closing and saving the document for each method rather than across the board. To do so, you can add columns to the aMethodsToApply array and move the opening and closing logic inside the main loop.

Using ProcessFiles

With the ProcessFiles method, we can write code to make changes to groups of documents or just to log various information. For example, the LogBugs method uses ProcessFiles to create a log of all occurrences of the "Bug" icon in a group of documents. It calls on the LogGraphics method discussed above. Here's the code:

```
* Create a log of files containing at least one bug icon.
* If LogGraphics has already been run on the set of
* files in question, uses that result. If not,
* runs it, then processes the results.
LOCAL nFileCount, nWorkArea
IF NOT USED("Graphics")
  * Set up an array and call ProcessFiles
  LOCAL aMeth[1,2]
  aMeth[1,1]="This.LogGraphics"
  aMeth[1,2]="1,'bug'"

  This.ProcessFiles(@aMeth, .T., .T., 0)
ENDIF
nWorkArea = SELECT()
SELECT Graphics
```

```
nFileCount = RECCOUNT()
* Now, if there are any, create a new cursor.
IF nFileCount > 0
  SELECT cDocument, COUNT(*) AS nBugCount ;
    FROM Graphics ;
    GROUP BY 1 ;
    INTO CURSOR Bugs
  nFileCount = _TALLY
ENDIF
SELECT (nWorkArea)
RETURN nFileCount
```

Note that this method passes 0 for the nSaveMode parameter to ProcessFiles because there's no reason to resave the documents in this case. Logging the graphics doesn't change the documents.

When we were doing the final preparation of the Hacker's Guide documents, we accumulated a whole list of things that needed to be done to each document before it was ready to go. It included zooming them back to normal mode, accepting and turning off revisions, performing a certain set of replacements (such as changing "VFP 98" to "VFP 6"), logging a number of other strings (such as "???", our indicator for notes to each other) so that we could check them manually, and checking for embedded graphics, since we were supposed to be using linked graphics. We used this program to "clean up" documents in a specified directory.

```
* Final clean-up for HackFox docs in Addns directory, including:
*  - zoom normal
*  - revisions off
*  - replacements
*  - log find strings
*  - log embedded graphics
#DEFINE HackHome "e:\Hack6\"
#DEFINE HackReady "Ready\Addns"
* Set up array of method calls
LOCAL aMeth[5,2]
aMeth[1,1] = "This.ZoomNormal"
aMeth[1,2] = ""
aMeth[2,1] = "This.RevisionsOff"
aMeth[2,2] = ".t."
aMeth[3,1] = "This.ReplaceText"
aMeth[3,2] = '.t.,"Replaced","test\ReplStrs.DBF"'
aMeth[4,1] = "This.ReplaceText"
aMeth[4,2] = '.f.,"Finds","test\FindStr.DBF"'
aMeth[5,1] = "This.LogGraphics"
aMeth[5,2] = "2"
* Now loop through all documents and create logs.
LOCAL oProcess
oProcess = NewObject("cusProcessDocuments","Process")
IF VarType(oProcess) <> "O"
  WAIT WINDOW "Sorry. No go."
```

```
  RETURN -1
ENDIF
WITH oProcess
  * Start Word
  .GetWord()
  * Handle each directory
  IF DIRECTORY(HackHome + HackReady )
    .cDirectory = HackHome + HackReady
    .cFileMask = "s" + "*.*"

    IF .GetFiles() > 0
      .ProcessFiles(@aMeth,.t.,.t.,2)
    ENDIF
  ENDIF
ENDWITH
RELEASE oProcess
RETURN
```

Final thoughts

The cusProcessDocuments class contains two methods to deal with the various log cursors it creates. LogToTable converts a log cursor to a table so that it can be kept for later review. LogReport produces a report (using VFP's quick report option) from a log cursor. While the reports generated this way are acceptable for developers, if you want to produce reports for end-users, you'll have to modify the results. (See Jeff Donnici's article in the April '99 FoxPro Advisor for ideas on how to manipulate an .FRX programmatically.) As an alternative, you can use Automation to produce the report in Word instead.

Making wholesale changes to large numbers of documents can seem overwhelming. But the cusProcessDocuments class presented here lets you automate the task, so that you can focus on what needs to be done instead of how to manage it.

The class contains a number of other methods not discussed in the article. The zReadMe method describes them all and the zScripts method shows how to use them. Enjoy.