Advanced Office Automation

Session ???

Tamar E. Granor, Ph.D.
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Phone: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com
Web: www.tomorrowssolutionsllc.com

Overview

Getting started automating Office isn't that hard. Doing the simple tasks with each of the servers is well-documented. But what comes next? This session will look at some more complex Automation tasks, including responding to Office events, shutting down abandoned servers, using Office's spelling checker, and more.

They’re all the same, right?

Since Office is a suite, if you’ve automated one of the servers, you might assume that working with the others is a breeze. In many ways, that’s true. Once you understand the nature of Automation, and know how to tease information out of the Office documentation, you’re ahead of the game.

But, there are some subtle variations (and some not so subtle) among the different applications that means code that works in one application might not work in another. To complicate matters further, while Word, Excel and PowerPoint have a lot in common, Outlook is different than the others in many ways.

Getting started

The first major difference has to do with instances and creation. Excel and Word let you create multiple instances, while PowerPoint and Outlook are restricted to single instances. This has several implications. First, if you call CreateObject() more than once for Excel or Word, you have more than one instance of the server running. However, with PowerPoint and Outlook, all the object references refer to the same instance. More significantly, if PowerPoint or Outlook is running before you call CreateObject(), the function returns a reference to the running instance.

Not surprisingly, if you create an instance of PowerPoint or Outlook via Automation and then start the same application through the Windows interface, there’s still only one instance running. More surprisingly, that’s true for Word as well. Excel, on the other hand, opens a separate instance in this situation.

Getting out

The differences continue when it comes to exiting the applications. When you shut down Word, PowerPoint or Outlook, the executable is unloaded, but when you close Excel, the executable stays in memory until you release the variable that held the reference. The object reference is still good, and you can talk to a number of its properties.

This has an impact on the code you need to check for a valid object reference to the server. For most of the Office servers, you can use code along these lines:

```
IF VARTYPE(oWord) = "O" AND TYPE("oWord.Name") = "C"
   * Server is running
ENDIF
```

But this code won’t work for Excel. The best solution I’ve found is to maintain an extra property or variable to track Excel’s expected visibility and test whether expected visibility matches actual visibility. (The issue is that, once you make Excel visible, a user might shut it down. As long as you keep it invisible, your application controls its existence and you don’t need to check whether your reference is valid, though a sophisticated user might use the Task Manager to close the application.) In this code, lShouldBeVisible is a variable; you could make it a property if you’re using a wrapper around the Excel server. Any code that affects Excel’s visibility needs to set lShouldBeVisible appropriately.

```
LOCAL lReturn

IF IsNull(oXL)
  * No instantiated server
  lReturn = .F.
ELSE
  * Compare actual Visible value to tracked visibility
  IF oXL.Visible = lShouldBeVisible
    * They match, so the server is open and good
    lReturn = .T.
   ELSE
    * Visibility doesn't match. User must have
    * shut server down.
    lReturn = .F.
  ENDIF
ENDIF

RETURN lReturn
```

Show me

If you've done much Automation work, you've undoubtedly learned that Automation code runs faster when the server is invisible. However, keeping PowerPoint invisible can be tricky. First, once you show PowerPoint by setting its Visible property to True, you can't hide it again. This command generates an error:

```
oPowerPoint.Visible = .F.
```

Second, a number of PowerPoint's properties, including ActivePresentation, aren't accessible unless PowerPoint is visible. To address individual presentations with PowerPoint hidden, you must use the Presentations collection.

What's in a template?

Another significant difference among the servers is the relationship of templates to documents. In Excel and PowerPoint, creating a new document based on a template simply makes a copy of the template as a starting point. In Word, a new document contains copies of the template's boilerplate text and styles, but retains a connection to the template's macros. This means that changes to the macro definitions in a Word template affect documents based on that template. That's not true for the other applications.

Go away!

When an application automates Office, it's not unusual to end up with abandoned instances of the various Office servers. This can happen due to bugs in your code, because of system problems, or for other reasons. Running a bunch of abandoned servers in the background can destabilize Windows and eventually lead to a crash.

Thus, it's handy to be able to search for and kill automation servers that aren't in use. While you can't do this through the Automation interface (after all, if you had an object reference to the server, you could simply call its Quit method), the Windows API provides the tools you need.

There are three steps in solving the problem:

1. Find each instance of the application.
2. Check whether it's visible.
3. If it's not visible, stop it.

The usual way to find a running application is with the FindWindow API function. You pass the window's title and the function returns a handle to the window. For example, this code finds the Calculator applet:

```
DECLARE LONG FindWindow IN WIN32API ;
   STRING lpClassName, STRING lpWindowName
nHandle = FindWindow( .NULL., "Calculator")
```

The problem with this approach is that you have to know the window's title. For the Office applications, that's not simple as the window title changes based on the open document.

Fortunately, there's another way to get to windows. Rather than looking for a particular window, it's possible to cycle through the list of open windows and check each to see whether it's one we're interested in. The main window for each running application is considered a child window of the Windows desktop, so the solution is to get a reference to the desktop and then look at its child windows. The GetDesktopWindow API function returns a handle to the Windows desktop, while the GetWindow function lets you cycle through all the child windows of a specified window. Here are the declarations:

```
DECLARE LONG GetDesktopWindow IN WIN32API
DECLARE LONG GetWindow IN WIN32API LONG hWnd, LONG wCmd
```

The parameters to GetWindow are the reference window and a number that indicates which window to return. Pass the constant GW_CHILD (5) to return the first child window of a window. Pass GW_NEXT (2) to get the next child; in this case, you pass the child window you already have as the first parameter. Putting this together, this code gets a reference to the desktop and then cycles through all its children:

```
#DEFINE GW_CHILD 5
#DEFINE GW_NEXT 2
lnDesktopHWnd = GetDesktopWindow()
lnHWnd = GetWindow( lnDesktopHWnd, GW_CHILD)
DO WHILE lnHWnd <> 0
   * Do something with the one you have here
   * Get the next one
   lnHWnd = GetWindow( lnHWnd, GW_NEXT)
ENDDO
```

Now that we can get a handle to each window, we need a way to check whether it's the main window of the application we're interested in. It turns out that each application has a class name that uniquely identifies it. For example, the class name for Word is "OpusApp"; for Excel, it's "XLMain" (which makes a lot more sense than Word's). The class name for PowerPoint depends on the version. For PowerPoint 2000, it's "PP9FrameClass". For PowerPoint 2002 (XP), it's "PP10FrameClass".

The GetClassName function looks up the class name for a window. Here's the declaration:

```
DECLARE LONG GetClassName IN WIN32API ;
   LONG hWnd, STRING lpClassName, LONG nMaxCount
```

The first parameter is the window handle. The second parameter must be passed by reference and holds the class name on return. The third parameter indicates the length of the string passed as the second parameter. The function returns the length of the string returned. Here's an example call:

```
lcClass = SPACE(256)
lnLen = GetClassName( lnHWnd, @lcClass, LEN(lcClass))
lcClass = LEFT(lcClass, lnLen)
```

Once we find an instance of the server application, we want to check whether it's visible. If so, then the user can shut it down as needed. If not, we'll shut it down. The IsWindowVisible API function takes a window handle as a parameter and returns 0 if it's hidden. Here's the declaration and call:

```
DECLARE LONG IsWindowVisible IN WIN32API LONG hWnd
lnIsVisible = IsWindowVisible( lnHWnd)
```

The last step in the process is to tell the application to shut down. The PostMessage API function handles that task. It accepts four parameters, but only two are needed in this case. (Pass 0 for the other two.) The first is the window handle and the second is the message to the window. In this case, the message to pass is WM_CLOSE (0x10). Here's the declaration and an example call:

```
DECLARE LONG PostMessage IN WIN32API ;
   LONG hwnd, LONG wMsg, LONG wParam, LONG lParam
#DEFINE WM_CLOSE 0x10
PostMessage( lnHWnd, WM_CLOSE, 0, 0)
```

We can put all this together to create a function that receives the class name of the application to shut down. Here's the code for KillApp:

```
FUNCTION KillApp
*==================================================
* Program:       KillApp.PRG
* Purpose:       Close any invisible instances of a specified program
* Author:         Tamar E. Granor
* Copyright:     (c) 2002 Tamar E. Granor
* Last revision: 04/16/02
* Parameters:   tcClassName - the classname of the app to close
* Returns:        Number of instances closed; -1, if parameter problems
*==================================================
#DEFINE GW_CHILD 5
#DEFINE GW_HWNDNEXT 2
#DEFINE WM_CLOSE 0x10

LPARAMETERS tcClassName

ASSERT VARTYPE(tcClassName) = "C" AND NOT EMPTY(tcClassName) ;
   MESSAGE "KillApp: Must pass class name of application to kill"

IF VARTYPE(tcClassName) <> "C" OR EMPTY(tcClassName)
```

```
   ERROR 11
   RETURN -1
ENDIF

DECLARE LONG GetDesktopWindow IN win32api
DECLARE LONG GetWindow IN WIN32API LONG hWnd, LONG wCmd
DECLARE LONG IsWindowVisible IN WIN32API LONG hWnd
DECLARE LONG GetClassName IN WIN32API LONG hWnd, STRING lpClassName,
LONG nMaxCount
DECLARE LONG PostMessage IN WIN32API LONG hwnd, LONG wMsg, LONG wParam,
LONG lParam

LOCAL lnDesktopHWnd, lnHWnd, lnOldHWnd, lcClass, lnLen, nClosedCount

lnDesktopHWnd = GetDesktopWindow()
lnHWnd = GetWindow( lnDesktopHWnd, GW_CHILD)
lnClosedCount = 0

DO WHILE lnHWnd <> 0
   lcClass = SPACE(256)
   lnLen = GetClassName( lnHWnd, @lcClass, 256)
   lnOldHWnd = lnHWnd
   lnHWnd = GetWindow(lnOldHWnd, GW_HWNDNEXT)
   IF UPPER(LEFT(lcClass, lnLen)) = UPPER(tcClassName)
      lnVisible = IsWindowVisible(lnOldHWnd)
      IF lnVisible = 0
         PostMessage( lnOldHWnd, WM_CLOSE, 0, 0)
         lnClosedCount = lnClosedCount + 1
      ENDIF
   ENDIF
ENDDO

RETURN lnClosedCount
```

If you regularly need to shut down a particular application, you can write a simple function that calls KillApp, passing the right class. For example, this function deals with instances of Word:

```
FUNCTION KillWord
*==================================================
* Program:       KILLWord.PRG
* Purpose:       Close any invisible instances of Microsoft Word
* Author:         Tamar E. Granor
* Copyright:     (c) 2002 Tamar E. Granor
* Last revision: 05/10/2002
* Parameters:   (None)
* Returns:        Number of instances closed

*==================================================
LOCAL lnKilled

lnKilled = KillApp("OpusApp")

RETURN lnKilled
```

For PowerPoint, you might choose to call KillApp with the class name for each version, so you don't have to worry about which version is installed. Finally, if you're interested in shutting down abandoned instances of all of the Office applications, you could use a function like this:

```
FUNCTION KillOfficeApps
*==================================================
* Program:        KILLOfficeApps.PRG
* Purpose:        Close any invisible instances of Microsoft Word,
*                 Excel, and PowerPoint 2000
* Author:          Tamar E. Granor
* Copyright:      (c) 2002 Tamar E. Granor
* Last revision:  05/10/2002
* Parameters:    (None)
* Returns:         Number of instances closed
*==================================================
LOCAL lnTotalKilled, lnKilled, aAppsToKill[3], lnPos

aAppsToKill[1] = "OpusApp" && Word
aAppsToKill[2] = "XLMain"  && Excel
aAppsToKill[3] = "PP9FrameClass"  && PowerPoint 2000

lnTotalKilled = 0

FOR lnPos = 1 TO ALEN(aAppsToKill, 1)
   lnKilled = KillApp( aAppsToKill[lnPos])
   lnTotalKilled = lnTotalKilled + lnKilled
ENDFOR

RETURN lnTotalKilled
```

The materials for this session include KillApp.PRG, which contains the KillApp function, as well as several application-specific functions.

A number of web articles list class names for various applications. Since websites change often, rather than listing specific articles here, I recommend searching for “class name” with “OpusApp”.

Can you spell that for me?

There are often situations where it would be useful to have the ability to check spelling in an application, but VFP doesn’t include a spelling checker. (There was a spelling checker application included with some versions of VFP, but it couldn’t be distributed with executable applications.) There are some third-party tools available, but if you know your users will have Office installed, you can build your own using Office’s spelling tool.

The Office spelling engine can’t be used as a stand-alone product, only from within the other Office tools. (This is both a license issue and a technical issue.) So, you need to check spelling using either Word or Excel, both of which expose spelling functionality. Word’s implementation is easier to automate and offers more functionality.

There are two ways to check spelling in Word. The CheckSpelling method of the application object accepts a string parameter and returns True or False to indicate whether the string passes the spelling check. So, if all you need is an indicator, code like this will work:

```
cString = "This is the string to chek."
oWord = CreateObject("Word.Application")
lIsCorrect = oWord.CheckSpelling(cString)
```

If you want recommendations on solving the problem, this approach is too simplistic. Instead, use the GetSpellingSuggestions method of the Application object. The method accepts a single word and populates a SpellingSuggestions collection with recommendations for fixing it. This approach requires at least one document to exist, though it can be empty. Here's an example:

```
cWord = "chek"
oWord = CreateObject("Word.Application")
oWord.Documents.Add()
oSuggestions = oWord.GetSpellingSuggestions( cWord )
```

To check a whole string, you can break it into words and pass each word in turn to the method, saving the returned results. I created a class called WordUtils (included in the materials for this session) with a CheckSpelling method. The class has three custom properties:

- oWord contains an object reference to Word.
- aSuggestions is an array property containing all the suggestions returned from the most recent spelling check.
- nSuggestions contains the total number of suggestions (the number of rows in aSuggestions).

The class also contains a method called CheckWord that ensures Word is running.

Here's the CheckSpelling method. This version works in VFP 7 and later. If you know you'll be using it only in VFP 8 and later, you might choose to change the aSuggestions array to a collection. To use it in VFP 6 and earlier, the code that breaks the string into words must be rewritten. It currently uses the GetWordCount() and GetWordNum() functions introduced in VFP 7; the FoxTools library offers alternate versions of these for earlier versions. (A few other syntactical changes are needed in VFP 6 and earlier.)

```
*==================================================
* Program:          CheckSpelling
* Purpose:          Use Word's spelling engine to check a string
* Author:           Tamar E. Granor
* Copyright:        (c) 2002 Tamar E. Granor
* Last revision:    02/15/02
* Parameters:       cString - the string to be checked
* Returns:          .T. if no spelling errors were found
*                   .F. if there were spelling errors or problems
*                   with params
*==================================================
LPARAMETERS cString

ASSERT VARTYPE(cString) = "C" ;
  MESSAGE "SpellCheck: First parameter (cString) must be character"

IF VARTYPE(cString) <> "C"
  ERROR 11
  RETURN .f.
ENDIF

LOCAL lReturn, nWords, nWord
LOCAL oSuggestions as Word.SpellingSuggestions
LOCAL oSuggestion as Word.SpellingSuggestion
```

```
DIMENSION This.aSuggestions[1]
This.aSuggestions[1] = ""
This.nSuggestioncount = 0

IF EMPTY(cString)
  lReturn = .t.
ELSE
  IF This.CheckWord()
    WITH This.oWord
      .Documents.Add()

      lReturn = .T.
      nWords = GETWORDCOUNT( cString)
      nSuggCount = 0
      FOR nWord = 1 TO nWords
        cWord = GETWORDNUM( cString, nWord )
        oSuggestions = .GetSpellingSuggestions( cWord )
        IF oSuggestions.Count <> 0
          lReturn = .F.
          * Parse the list and put into the array
          FOR EACH oSuggestion IN oSuggestions
            This.nSuggestionCount = This.nSuggestionCount + 1
            DIMENSION This.aSuggestions[ This.nSuggestionCount, 3]
            This.aSuggestions[ This.nSuggestionCount, 1 ] = nWord
            This.aSuggestions[ This.nSuggestionCount, 2 ] = cWord
            This.aSuggestions[ This.nSuggestionCount, 3 ] = ;
              oSuggestion.Name
          ENDFOR
        ENDIF
      ENDFOR
    ENDWITH
  ELSE
    lReturn = .F.
  ENDIF
ENDIF

RETURN lReturn
```

The class also contains code in the Destroy method to shut down the Word instance. As written, the class opens Word once and keeps it open. This means that the first spelling check may be a little slow, but after that, it should be fast. Here's an example of using the class:

```
cString = "This is the string to chek"
oSpeller = NewObject("cusSpellCheck", "WordUtils")
IF NOT oSpeller.CheckSpelling(cString)
  * Do something about misspellings
ENDIF
```

The materials for this session include a form (Spelling.SCX) that demonstrates the use of the spelling class.

Outlook is different

Despite their many differences, automating one of Word, Excel or PowerPoint is pretty much like automating the others. Although Outlook is a member of Office, it works differently in many ways. The most significant is in working with Outlook's collections.

In the other Office applications, it doesn't matter whether you work with a collection directly through its owner or save a reference to a local variable—the results are the same (though the local variable can speed things up). In Outlook, that's not necessarily true. When you refer to a collection in Outlook, internally, a copy of the collection is made and a reference to it returned. This means that any changes you make happen on the copy, not the original. If you then look at the original, you won't see your changes. For example, consider this code:

```
oOutlook = CreateObject("Outlook.Application")
oNS = oOutlook.GetNameSpace("MAPI")
* Get tasks
oFolder = oNS.GetDefault(13)
FOR EACH oItem IN oFolder.Items
  ?oItem.Subject
ENDFOR
* Now sort it
oFolder.Items.Sort("Subject")
FOR EACH oItem IN oFolder.Items
  ?oItem.Subject
ENDFOR
```

When you run this code, the list prints out in the same order each time. That's because the Items collection is copied when it's referenced, and the Sort method is applied to the copy. To make this code work, you have to save the sorted copy and traverse it, like this:

```
oOutlook = CreateObject("Outlook.Application")
oNS = oOutlook.GetNameSpace("MAPI")
* Get tasks
oFolder = oNS.GetDefault(13)
* Now sort it
oItems = oFolder.Items
oItems.Sort("Subject")
FOR EACH oItem IN oItems
  ?oItem.Subject
ENDFOR
```

Working with document properties

All of the Office applications offer properties for their documents and let you define custom properties. In Word, Excel and PowerPoint, the Properties dialog shows both built-in and custom properties. In Outlook, properties show up in various places, depending on the type of item.

For Automation purposes, Word documents, Excel workbooks and PowerPoint presentations all have two properties, BuiltinDocumentProperties and CustomDocumentProperties, which point to DocumentProperties collections. The two DocumentProperties collections contain DocumentProperty objects. Both the DocumentProperties collection and the DocumentProperty object are Office objects, so

documentation is found in the “Microsoft Office Visual Basic Reference” section of the various Help files.

The various item objects in Outlook each have a UserProperties collection, which is based on a different object than DocumentProperties. However, it’s still possible to add properties and to check the values of existing properties.

Why would you want to work with these collections? For a variety of reasons. For the applications other than Outlook, BuiltinDocumentProperties is where you find information like the author, the time the document was created, the subject, and so forth. Custom properties let you add information, for example, to identify documents created by your application.

Reading and changing the built-in document properties is a little tricky. The properties themselves are objects, so you have to look at the Value property of a document property to see what it contains. For example, to check the title of a Word document accessed through oDocument, you can use:

```
#DEFINE wdPropertyTitle 1
cTitle = oDocument.BuiltinDocumentProperties( wdPropertyTitle ).Value
```

or you can break it into two steps:

```
oTitleProp = oDocument.BuiltinDocumentProperties( wdPropertyTitle)
cTitle = oTitleProp.Value
```

To change a built-in property, assign a new value to the Value property (though some are read-only). For example, to set the author of an Excel workbook accessed through oWorkbook, use:

```
oWorkbook.BuiltinDocumentProperties("Author").Value = cAuthor
```

Again, if you prefer, you can do it in two steps:

```
oAuthorProp = oWorkbook.BuiltinDocumentProperties("Author")
oAuthorProp.Value = cAuthor
```

To add a property to a Word, Excel or PowerPoint document, use the Add method of the CustomDocumentProperties collection. To add a property and specify its value, you need the first four parameters to this method. The first parameter is the property name, the third is the type, and the fourth is the value. Pass .F. for the second parameter. For example, to add a property called iDocumentID to a Word document and set its value to the variable m.iID, use code like this:

```
#DEFINE msoPropertyTypeNumber 1
oDocument.CustomDocumentProperties.Add( ;
  "iDocumentID", .F.,  msoPropertyTypeNumber , m.iID)
```

Custom properties added in this way show up on the Custom page of the Properties dialog for the document.

Once you’ve added properties, you probably want to be able to read them by Automation, as well. Unfortunately, there’s no direct way to check whether a particular property exists. You have to use the brute force approach:

```
lFound = .F.
FOR EACH oProp IN oDocument.CustomDocumentProperties
  IF UPPER(oProp.Name) == "IDOCUMENTID"
    * Found it. Get out of here
    lFound = .T.
    EXIT
  ENDIF
ENDFOR

IF lFound
  iDocID = oProp.Value
ENDIF
```

For Outlook, you do things similarly, but not quite the same. To add a custom property to an item, use the Add method of the UserProperties collection. Two parameters are required: the name of the property and its type. The value is set separately. For example, to add a property to a contact item referenced through oContact, you'd use code like this:

```
#DEFINE olText 1
oContact.UserProperties.Add("AIMName", olText)
oContact.UserProperties("AIMName").Value = "TeddyBear"
```

By default, when you add a custom property to an item, all items of that type get the new property. (In the example above, every contact gets an AIMName property.) The Add method takes an additional parameter to turn that feature off. Pass .F. for the third parameter to indicate that this property should be added only to this item.

You don't need brute force to check for a custom property in Outlook. The Find method of the collection searches for a specified property and returns it as an object, if it's there. For example:

```
oAIMName = oContact.Find("AIMName")
IF VARTYPE(oAIMName) <> "X"
  cAIMName = oAIMName.Value
ENDIF
```

Responding to Office events

While most Automation code simply issues commands to the various Office applications to create, edit, parse or print documents, at times, you need to respond to things that happen in the server application. For example, when automating Outlook, you might present the user with the bare bones of a mail message and allow him to complete it. When the user saves the message, you want your application to do something. In one application I worked on, all documents were tracked in a VFP database and each time a user finished working with a document, we logged the user's name and the time.

There are two approaches to dealing with Office events. The first is useful only when you know that your VFP application will be running when the user is working in Office. The second can be used with no VFP application running, as well.

Binding server events with EventHandler()

The first solution uses the EventHandler() function added in VFP 7. To use this approach, you need to create a class that implements the appropriate interface of the server

application. An interface is a set of methods defined by a server. Implementing an interface means providing code for those methods. It's similar to inheriting from a class, but the class that implements an interface doesn't execute any code from the original class. In addition, you can implement an interface written in a different language. The IMPLEMENTS keyword was added to DEFINE CLASS in VFP 7.

How do you know what interface you need to implement? The Object Browser lets you explore the interfaces of an Automation server. Open the appropriate type library and expand the Interfaces node to see all the interfaces supported. Table 1 shows the main interface that contains events you can respond to for each of the Office servers. (Some of the applications have more than one interface containing events.)

Table 1. The various Office servers each support one or more interfaces that let you respond to events. The principal interface for each application is shown here.

Server	**Interface**	**Methods**
Excel	AppEvents	NewWorkbook, SheetActivate, SheetBeforeDoubleClick, SheetBeforeRightClick, SheetCalculate, SheetChange, SheetDeactivate, SheetFollowHyperlink, SheetSelectionChange, WindowActivate, WindowDeactivate, WindowResize, WorkbookActivate, WorkbookAddinInstall, WorkbookAddinUninstall, WorkbookBeforeClose, WorkbookBeforePrint, WorkbookBeforeSave, WorkbookDeactivate, WorkbookNewSheet, WorkbookOpen
Outlook	ApplicationEvents	ItemSend, NewMail, OptionsPagesAdd, Quit, Reminder, Startup
PowerPoint	EApplication	NewPresentation, PresentationClose, PresentationNewSlide, PresentationOpen, PresentationPrint, PresentationSave, SlideShowBegin, SlideShowEnd, SlideShowNextBuild, SlideShowNextSlide, WindowActivate, WindowBeforeDoubleClick, WindowBeforeRightClick, WindowDeactivate, WindowSelectionChange
Word	ApplicationEvents2	DocumentBeforeClose, DocumentBeforePrint, DocumentBeforeSave, DocumentChange, DocumentOpen, NewDocument, Quit, WindowActivate, WindowBeforeDoubleClick, WindowBeforeRightClick, WindowDeactivate, WindowSelectionChange

Once you're looking at the right interface in the Object Browser, you can expand its Methods node to see the supported events. Figure 1 shows the Word ApplicationEvents2 interface expanded in the Object Browser.

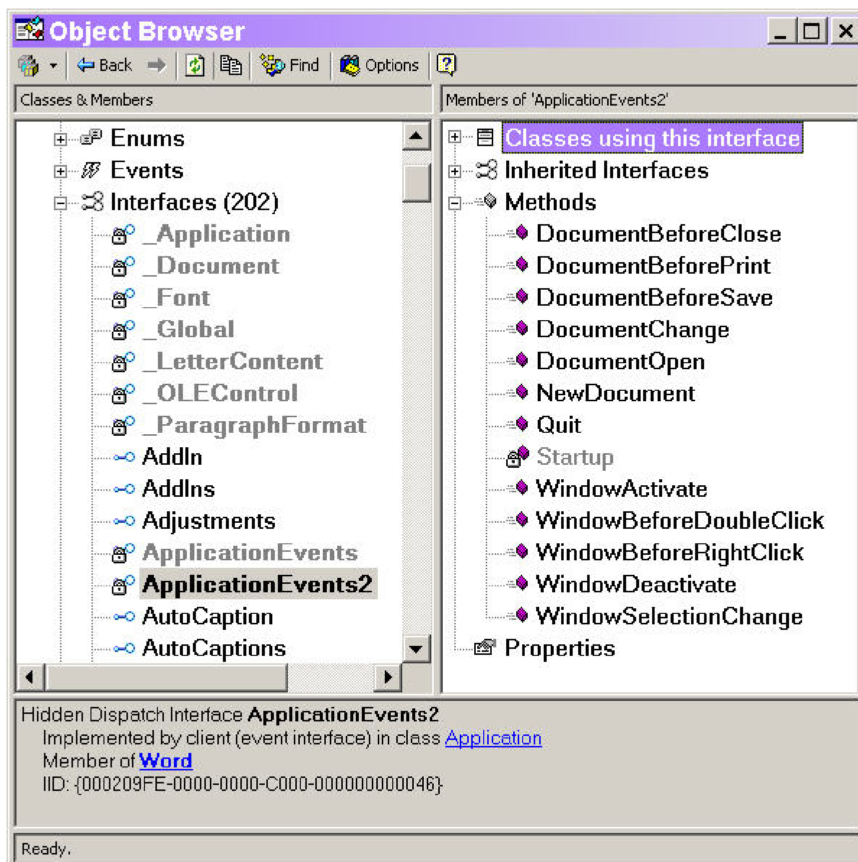

Figure 1. The Object Browser lets you explore the interfaces supported by a server. You can drag an interface to a code window to create a class that implements the interface.

The Object Browser does more than just let you find out what interfaces are available and what methods they support. Drag an interface to an editing window (MODIFY COMMAND) and a class is defined that implements that interface. For example, if you drag Word's ApplicationEvents2 interface to a code window, this code is generated. (The code here has been slightly reformatted to fit the page.)

```
x=NEWOBJECT("myclass")

DEFINE CLASS myclass AS session OLEPUBLIC
   IMPLEMENTS ApplicationEvents2 IN ;
     "c:\program files\microsoft office\office\msword9.olb"

   PROCEDURE ApplicationEvents2_Quit() AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_DocumentChange() AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_DocumentOpen(Doc AS VARIANT) AS VOID
   * add user code here
   ENDPROC
```

```
   PROCEDURE ApplicationEvents2_DocumentBeforeClose(Doc AS VARIANT, ;
     Cancel AS LOGICAL) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_DocumentBeforePrint(Doc AS VARIANT, ;
     Cancel AS LOGICAL) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_DocumentBeforeSave(Doc AS VARIANT, ;
     SaveAsUI AS LOGICAL, Cancel AS LOGICAL) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_NewDocument(Doc AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_WindowActivate(Doc AS VARIANT, ;
     Wn AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_WindowDeactivate(Doc AS VARIANT, ;
     Wn AS VARIANT) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_WindowSelectionChange(Sel AS VARIANT) ;
     AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_WindowBeforeRightClick(Sel AS VARIANT, ;
     Cancel AS LOGICAL) AS VOID
   * add user code here
   ENDPROC

   PROCEDURE ApplicationEvents2_WindowBeforeDoubleClick(Sel AS VARIANT,;
     Cancel AS LOGICAL) AS VOID
   * add user code here
   ENDPROC

ENDDEFINE
```

To create an event handler, put code in the methods for the events you want to handle. Do not delete the other methods. For a class to implement an interface, every method of the interface must be included.

It's a good idea to change the reference to the class in the IMPLEMENTS line. The Object Browser generates a reference to the type library, including the path. If your event handler will be used on more than one machine, the path may be wrong. It's better to use the application's ProgID (such as "Excel.Application"). Unfortunately, Word doesn't support using the ProgID; in that case, your best bet is to use Word's GUID followed by version information, like this:

```
IMPLEMENTS applicationevents2 in ;
```

```
{00020905-0000-0000-C000-000000000046}#8.1
```

You probably want to change the name of the class, as well.

Once you've defined the event handler class, you can bind it to an instance of the server using the EventHandler() function. This example binds the class shown above to a Word instance.

```
oWord = CreateObject("Word.Application")
oHandler = CreateObject("MyClass")
EVENTHANDLER( oWord, oHandler)
```

The bindings last as long as the objects involved stay in scope. Alternatively, you can unbind the handler from the server by calling EventHandler() again and passing .T. for the optional third parameter.

The session materials include a class (OfficeEventHandler9.PRG) that implements the principal interfaces of Word, Excel and PowerPoint. For each, it logs opening, closing, saving and creating a new document to a table. The materials also include a form (OfficeEvents9.SCX) to open the servers and display the log. (VFP 8 has a bug that prevents you from implementing the Excel and PowerPoint interfaces with a single class. The conference materials include VFP 8 versions, OfficeEventHandler.PRG and OfficeEvents.SCX, that work around the bug. Unfortunately, the result is far more complex code for instantiating and keeping track of the event handlers.)

Binding server events with an add-in

The second approach is more complicated. It requires VBA code to create what's known as an *add-in* and a VFP COM object to be used by the add-in. It's useful when you can't be sure that VFP will be running at the time the server event occurs.

An add-in is a special document in Word, Excel or PowerPoint that contains code. (While Outlook also supports add-ins, the mechanism is different enough that it's not discussed here.) Add-ins can be loaded interactively or via Automation.

An event handler add-in needs two components. There's a class that contains the actual event handling code, plus separate code to connect the event handler class to the application object (just as EventHandler() does in the VFP example). Both of these pieces are written in Visual Basic for Applications (VBA) using the Visual Basic Editor (VBE), which is available in all of the Office applications. While the particulars, especially the event methods available, vary from one Office server to the next, the general structure is the same for each.

Creating an add-in

To create an add-in, create a new, blank document in the server application and then open the VBE by choosing Tools | Macro | Visual Basic Editor from the menu. The first step in the VBE is to create the event handler class. To do so, make sure the Project Explorer is open. It's normally docked on the left-hand side of the VBE underneath the menu. If it's not open, use View | Project Explorer to open it. Right-click on the project for your

document and choose Insert | Class Module. Figure 2 shows the Project Explorer in Word after adding the new class module.

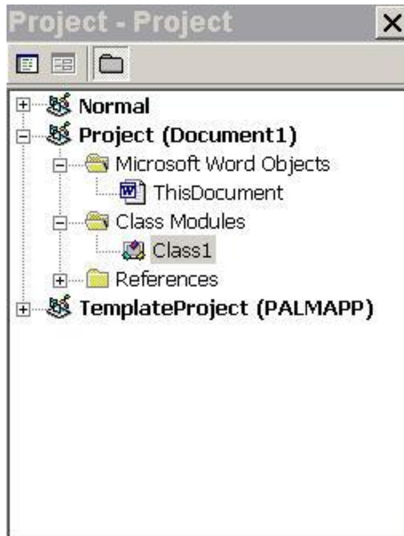

Figure 2. The Project Explorer provides an overview of your VBA project. Here, the class module for the event handler has been added.

When you add the class module, a code window opens to hold code for that module. That's where the event handler code goes.

Before starting to write code, it's a good idea to rename the event class. Click on the new class module in the Project Explorer. Then, in the Properties window (normally docked below the Project Explorer), click into the Name property and give your class a meaningful name, like EventHandler.

To create an event handler for one of the Office servers, you need an object reference to the application object. To add one, declare a property to hold the reference. Switch to the code window, making sure the dropdowns show "(General)" and "(Declarations)," respectively, and type this code:

```
Public WithEvents WordApp As Application
```

The WithEvents keyword lets the object respond to events. Once you've completed this line and pressed Enter, the dropdowns at the top of the code window change. The left-hand dropdown, which shows the available objects, now includes your application property as in Figure 3. When you choose the property in the left-hand dropdown, the right-hand dropdown (Figure 4) is populated with the events to which your code can respond.

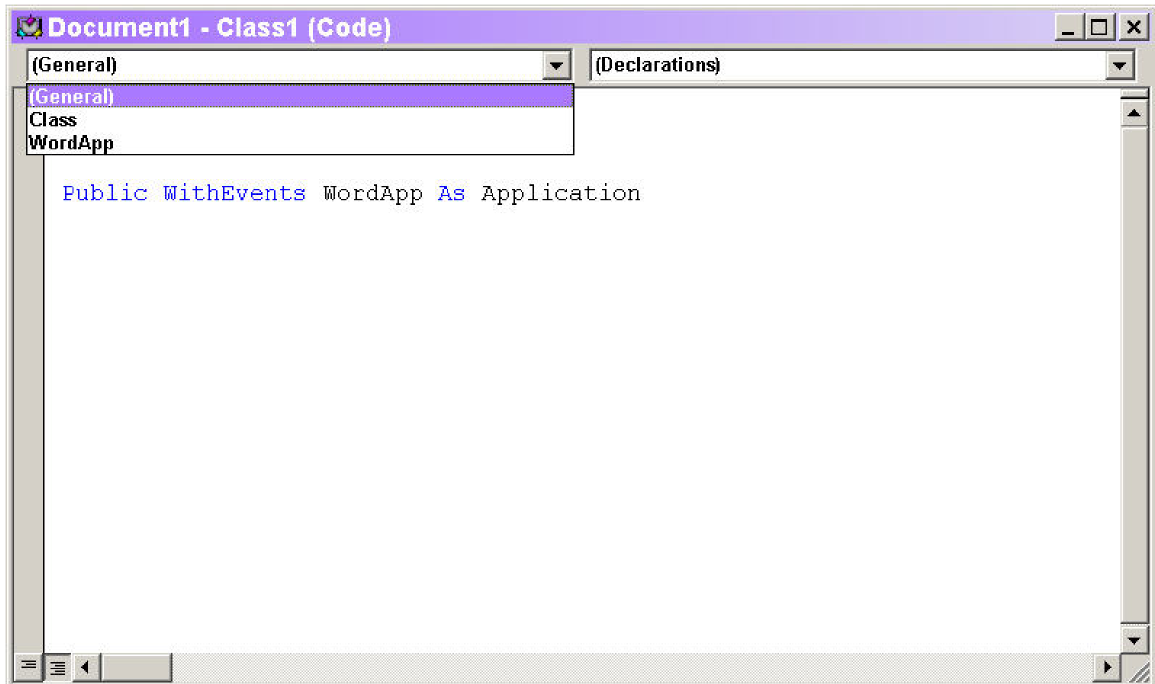

Figure 3 Once you declare a variable to hold the application object, that object is accessible in the code window.

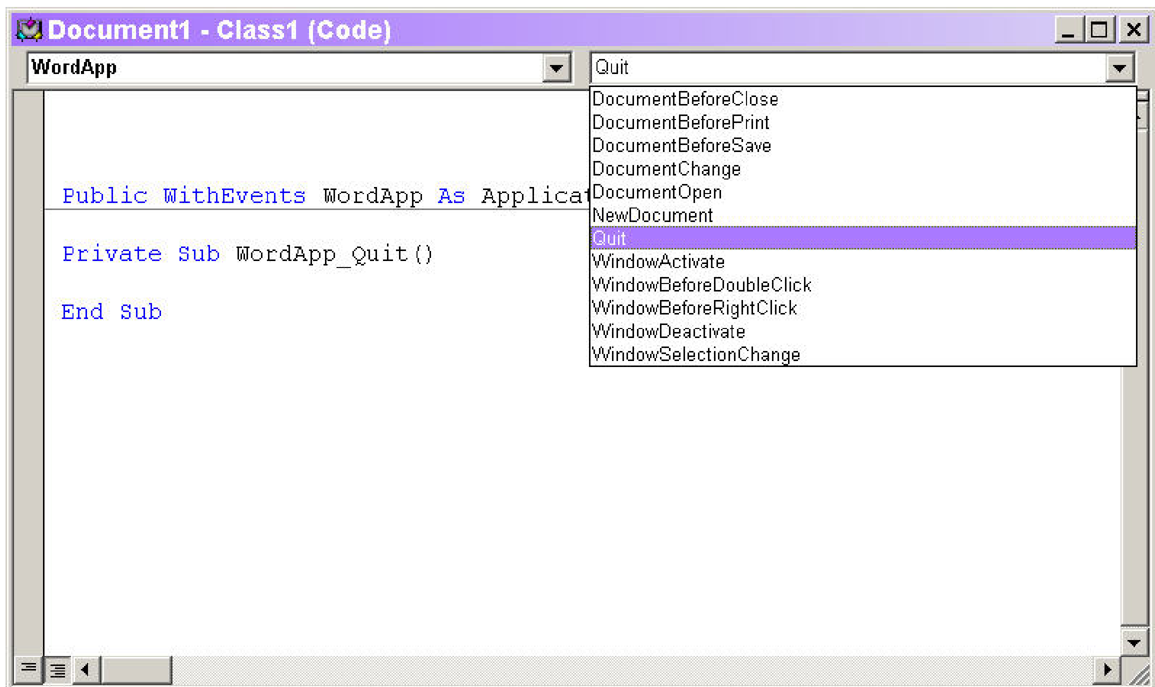

Figure 4 When you choose the application property from the objects dropdown, the right-hand dropdown shows the list of available events.

When you choose an event from the right-hand dropdown, stub code for that event is inserted into your code window. (In Figure 4, you can see the stub for the Quit event.)

Choose each event for which you want to write code from the events dropdown and add the VBA code that should run in response to that event. For example, this code responds to the NewDocument and DocumentBeforeClose events, in both cases displaying a message box that tells the user what's going on.

```
Public WithEvents WordApp As Application

Private Sub WordApp_DocumentBeforeClose(_
  ByVal Doc As Document, Cancel As Boolean)
MsgBox ("About to close " & Doc.FullName)
End Sub

Private Sub WordApp_NewDocument(ByVal Doc As Document)
MsgBox ("New document opened")
End Sub
```

Next, you need to connect the actual application object to the defined object reference property (WordApp, in the example). Add a module (not a class module) to the project. Right-click on the project in the Project Explorer and choose Insert | Module. As before, rename the new module by choosing it in the Project Explorer and changing the Name property in the Properties window. (I call mine "CatchWord".)

In this new module, declare a variable as a new instance of the event handler class you just created:

```
Dim cWordHandler As New EventHandler
```

Add a method whose sole purpose is to connect the application property of the event handler to the application object. I call it HandleEvents. Here's the Word version of the code:

```
Sub HandleEvents()
Set cWordHandler.WordApp = Application
End Sub
```

Next, we need a way to make this code run every time the add-in is loaded. While each of the applications has an event that fires automatically when a document or the application is opened, those events behave differently in different circumstances. (For example, Word doesn't run its AutoExec method when Word was started by automation.) So it's better to use code to explicitly run HandleEvents after loading the add-in. That code is included in "Loading the Add-in" below.

The document now has everything it needs to handle events. Next, save it as an add-in. The technique varies with the application. In all cases, though, switch back to the main application (rather than the VBE) before choosing Save As.

For Word, simply save the document as a template. If you store it in Word's startup folder (as specified on the File Locations page of Word's Options dialog), the add-in loads automatically each time you start Word. Alternatively, you can store it elsewhere and load it manually or via Automation each time you want it.

With Excel and PowerPoint, you need to save the document twice because you can't open an add-in directly for editing. First, save it as the native format for the application (.XLS for Excel, .PPT for PowerPoint), then use Save As to save it again as an add-in. If you

need to modify it later, open the native format version, make the changes, then save it both in the native format and as an add-in.

To save as an add-in in Excel, choose "Microsoft Excel Add-In (*.xla)" from the "Save as type:" dropdown. Once you make this choice, the dialog switches to the AddIns directory for the current user. Saving your add-in in that directory puts it on the list of add-ins available from Excel's Add-Ins dialog, but doesn't automatically enable it. You have to enable it in that dialog or via Automation. If you save it elsewhere, you can load it using the Add-Ins dialog or via Automation. (If you save it in the user's XLStart directory, it's loaded automatically.)

Saving an add-in in PowerPoint is pretty much the same as for Excel. Once you've saved the presentation, use File | Save As and choose "PowerPoint Add-In (*.ppa)." Again, the dialog switches to the AddIns directory. However, saving your add-in there doesn't have any special advantages except that the dialog for loading an add-in defaults to looking in that directory. Regardless of where it's saved, you can load an add-in through the Add-Ins dialog or via Automation.

Loading the add-in

There are several ways to get your add-in running. In general, it's a two-step process. First, add-ins need to be "registered" to make the application aware of them. Once registered, an add-in can be loaded to get it running. In some cases, you can do both steps at once.

You get the most control by using Automation code to load your add-ins. Each of the servers has an AddIns collection with an Add method. The exact behavior of the Add method varies, however, and, in Excel, using AddIns.Add isn't the best choice.

The Add method of Word's AddIns collection both registers and loads the add-in. All you need to do afterward is run the HandleEvents method to create the connection to the application object:

```
oWord = CreateObject("Word.Application")
oWord.AddIns.Add("WordEventHandler.DOT") && Add path
oWord.Run("HandleEvents")
```

In PowerPoint, the Add method registers the add-in, and you set the Loaded property to True to load it. Then run the HandleEvents method to create the connection.

```
oPPT = CreateObject("PowerPoint.Application")
oAddIn = oPPT.AddIns.Add("PPTEventHandler") && Add path
oAddIn.Loaded = .T.
oPPT.Run("HandleEvents")
```

Excel doesn't let you register and load an add-in with AddIns.Add unless there's an open workbook. In addition, when you leave an add-in loaded when Excel closes, the next time you run Excel, the add-in shows as installed, but doesn't actually get properly loaded. A better approach with Excel is to open the add-in with the Workbooks.Open method. That registers and loads the add-in cleanly. As with the others, you then need to run the HandleEvents method.

```
oXL = CreateObject("Excel.Application")
```

```
WITH oXL
   * Add path in next line
   oWorkbook = .Workbooks.Open("ExcelEventHandler.XLA")
   .Run("HandleEvents")
ENDWITH
```

Add-ins for Word, Excel and PowerPoint that log creation, opening, saving and closing of documents to a VFP table are included in the session materials.

Talking to VFP via a COM object

The final piece of this approach is a way to communicate with FoxPro. Depending what you want, there are a couple of possibilities. The VBA code can use ADO to update VFP data directly. Alternatively, the VBA code can instantiate a VFP COM object. We'll look at the second method here.

To create a COM object in VFP, you need a class defined as OLEPUBLIC, and a project containing the class. In the class, put methods for whatever should be done when the Office events fire. My experience is that it's best to have very granular methods, each performing a single, fairly simple task. In one application, I used one method to update a document log to indicate that the specified document was edited. That application used a semaphore locking scheme for the documents (to allow only one user to edit a document at a time), so another method released the semaphore lock when the document was closed.

The Session class is designed to be used for COM objects; it's extremely lightweight. The downside is that Session classes can't be created in the Class Designer; you have to write code in a PRG file.

One thing you probably want in all COM objects is some kind of error handler. Since most COM objects can't have a user interface, a good way to deal with errors is to log all relevant information to a text file (or a table). In the example here, the Error method handles any errors. If your COM object calls on other objects or outside code, and you want unified error handling, you're better off using ON ERROR to set up a global error handler. (Of course, an ON ERROR handler won't be called by any object that has its own error handling code.)

Here's a simple class (OfficeResponder.PRG in the session materials) that has one method to update a log table. The Init method ensures that the log table exists.

```
DEFINE CLASS OfficeResponder AS Session OLEPUBLIC
* COM server to be called in response to Office events.

PROTECTED cLogTable, cLogPath, cLogFullPath
cLogTable = "OfficeLogFromAddIn.DBF"

PROTECTED PROCEDURE Init
* Set up

This.cLogPath = SYS(2023)
This.cLogFullPath = FORCEPATH(This.cLogTable, ;
                              This.cLogPath)

* Make sure the log table exists
IF NOT FILE(This.cLogFullPath)
```

```
    CREATE TABLE (This.cLogFullPath) ;
        (cDocument C(60), cAction C(12), tModified T)
    CLOSE TABLES
ENDIF

SET EXCLUSIVE OFF
RETURN

ENDPROC

PROCEDURE UpdateLog( cDocument as String, cAction as String) as Boolean
* Update the activity log

IF VARTYPE(cDocument) <> "C" OR EMPTY(cDocument)
    ERROR 11
    RETURN .F.
ENDIF

IF VARTYPE(cAction) <> "C" OR EMPTY(cAction)
   ERROR 11
   RETURN .f.
ENDIF

INSERT INTO (This.cLogFullPath) ;
   VALUES (m.cDocument, m.cAction, DATETIME())

RETURN .t.

ENDPROC

PROCEDURE Error(nError, cMethod, nLine)

LOCAL lcErrorMsg, lcFileName

lcErrorMsg = "Error " + TRANSFORM(nError) + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Method " + cMethod + SPACE(1) + ;
             CHR(13) + CHR(10) + ;
             "Line " + TRANSFORM(nLine) + SPACE(1) + ;
             CHR(13) + CHR(10)+ ;
             "At " + TRANSFORM(DATETIME())

************************************************
* Dump an error log into the specified directory
************************************************
lcFileName = FORCEPATH("OfficeResponder.ERR", ;
                       This.cLogPath)
STRTOFILE(lcErrorMsg, lcFileName, .T.)
LIST MEMORY TO FILE (lcFileName) ADDITIVE noconsole
LIST STATUS TO FILE (lcFileName)  ADDITIVE noconsole
RETURN
ENDPROC

ENDDEFINE
```

To turn the class into a COM object, create a project (also called OfficeResponder in the example and included in the session materials), add the class to it, and build a .DLL COM server. The resulting .DLL file contains the COM object.

To use the COM object from the Office event handler code (that is, in VBA code), declare an appropriate variable, then use CreateObject() to instantiate the server. Call its methods as needed, and when you're done, set the object variable to Nothing (VBA's version of .null.) to release the server.

In this example (from WordLogger.DOT in the session materials), Word's DocumentBeforeClose method instantiates the OfficeResponder object and calls the UpdateLog method:

```
Private Sub wordapp_DocumentBeforeClose(ByVal Doc As Document, Cancel As Boolean)

Dim oVFPResponder, lResult As Boolean

Set oVFPResponder = CreateObject("OfficeResponder.OfficeResponder")
lResult = oVFPResponder.UpdateLog(Doc.FullName, "Closed")
Set oVFPResponder = Nothing

End Sub
```

In Excel, if you load the add-in as described above, the WorkbookBeforeClose code fires when you close the workbook that contains the add-in (such as when you close Excel), so you'll get a record in the log for the add-in itself.

Make sure that your main VFP application registers and loads the appropriate add-in at the same time that it instantiates each of the Office applications.

Summing Up

The Office servers are remarkably capable. Learning to do more than just create documents, format them, print them and save them will give you tremendous possibilities for application development.

Thanks to Ted Roche and John Hosier, who helped me develop some of the ideas here, and to my Advisor Answers co-columnists, Christof Lange and Pamela Thalacker, who reviewed some of this material in an earlier form.

Copyright, 2004, Tamar E. Granor, Ph.D.