THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Chapter 3

Document Generator

Automating document creation using Word, Outlook and a COM Add-In

In this chapter we'll try and illustrate how to combine Word and Outlook to automate the document creation process. By wrapping all logic into a COM add-in hosted within Outlook, the Document Generator is readily available to churn out documents in the blink of an eye. To build this Document Generator, you should be familiar with:

* The Word Object Model
* The Outlook Object Model
* The COM Addin IDTExtensibility2 Interface
* The .Net Framework

In this chapter we will cover:

* **The Business Scenario**: This section provides context for our application within our example vehicle: Bravo Corp. And shows some of the drivers behind a typical lightweight use case
* **Designing the Document Generator:**
* **Laying the Foundation:** This section discusses the prerequisites we must address before we can put our design into production
* **Creating the Add-In:** Here we'll step through how create the Document Generator add-in by easily harnessing features within Word and Outlook to help us.

The Business Scenario

A Bravo Corp Account Managers' job is to communicate all day, every day with event team members (both Bravo staff and the client). The role of the Account Manager (AM) is to be the intermediary between the client and his own events team. AMs work with "clients" at all levels (e.g. both the organization hosting a grand conference and any of the exhibitors simply taking a booth) to set their expectations and help define their needs.

AMs are senior level managers primarily responsible for managing all events for larger clients. A larger client is typically an organization that hosts several large trade shows or similar type events each year. On rare occasions, an AM is assigned to a client who hosts a single event each year (an event in size and stature similar to Comdex or Tech-Ed)

The Issue

In recent years, the AMs have consistently voiced the need for a tool allowing for the automation of document creation. As with any project, event management generates

hundreds of documents. From Service Contracts to Work Authorizations, Exhibit Hall Layouts to Booth Designs, an account manager spends each day shuffling paper and communicating with both the project team and the client.

Knowing that each document is different from the next, the prevailing thought was to automatically insert key pieces of data (i.e. client information) into the document and free the account manager to focus on other value added details. Not only would such a system save time, it would also reduce errors and, in the long run, increase customer satisfaction and loyalty.

The Solution – An Overview of the Document Generator

The book version of the Document Generator is scaled down and focuses on a single scenario. For our purposes, we will work under the scenario in which an Account Manager has just received news that Bravo Corp has won a new account for event management services. As Figure 3-1 illustrates, the scenario begins with an account manager receiving notification that a new account has been won.

Insert 0197f0301.tif

Figure 3-1. The Document Generator Architecture

With Microsoft Outlook open, the account manager initiates the document creation process by selecting Bravo Tools➢Create Project Documents from the menu. A form is displayed which lists all available project documents. These documents have been previously downloaded from the Bravo Corp server and stored on the user's system in the same manner as the Presentation Generator (see Chapter 2). The account manager selects the desired document or documents they wish to create. In order to insert client information into the documents, 'client records' is also selected. The client records listed in the combo box are pulled from the Microsoft Outlook contacts folder specified in the add-in's user settings.

Each document selected is opened by the Document Generator to determine if any Bookmarks exist (Bookmarks are great and I will teach you a bit more about them later on in this chapter). If any bookmarks exist within the Word document, a form is displayed listing each bookmark. Using the selected contact record, the form is pre-populated with the client information.

Once the user is satisfied with the information to be inserted into the document (or documents), the Document Generator inserts the client information and saves the documents to the "Save" folder specified in the user settings file. The process is then completed with the creation of an Outlook email item containing each of the newly created documents as attachments. The `To:` field of the email is populated using the client's email address and the `Attachments` field contains a listing of the attached documents. All that is left for the account manger to do is to review the documents, add some content to the body of the email, and hit `Send`.

> NOTE: Yes, I know I could have (and one could argue should have) added some functions that encapsulated in code a more sophisticated editing feature, but in the end, I

felt like that would be much too much work for little benefit. Instead, I decided to make sure the users knew about the "cheap and easy" double-click feature for opening a document in its application for editing. Simple and affective

Designing the Document Generator

The Document Generator is a COM Add-in hosted within Microsoft Outlook. Like the Presentation Generator, the Document Generator is a rich client experience utilizing the features of Microsoft Office and the Windows operating system. The add-in is the main component of the application and handles all interaction with the other components that comprise the add-in solution (including the user!).

The add-in is composed of five components as follows (Figure 3-2):

* **Microsoft Outlook**: The host application for the COM add-in and primary user interface.
* **Document Generator COM Add-in**: The dynamic link library containing the class implementing the IDTExtensibility2 interface and all other classes required for the add-in to function.
* **Microsoft Word**: Used by the COM add-in to control Word Documents.
* **Microsoft SQL Server**: Published Word Document templates are stored within a table of the Bravo Corp database. This table stores the actual binaries of the templates and its contents are available for a user to download at any time.
* **Word Documents**: Word documents are the product of the add-in. Each document contains bookmarks specify text placeholders for inserting content specific to a client.

Word Bookmarks

Word provides several ways to insert text into a document, the Content, Range and Selection objects being the most obvious. However, Word provides a perfect object for inserting text placeholders within a document, the Bookmark object. Like bookmarks in the physical world, Word bookmarks mark a placeholder in a document for future reference.

In the case of the Document Generator, bookmarks are inserted into every published Word template by the Bravo Documents Team to mark insertion points for client data. Where possible, the bookmark names match up with the properties of an Outlook ContactItem. For example, the company field of a ContactItem form in Outlook is named "CompanyName". Any document that needs to automatically include the company name of a selected contact will have a bookmark named "CompanyName". You will learn how this works later in the chapter.

Caution: This strategy certainly isn't bullet proof so please no nasty emails! When utilizing this strategy of mapping Bookmark names to ContactItem properties, just be aware of the fact that

changing Bookmark names can cause you to easily get out of sync and waste all kinds of time debugging. If at any time data is not inserting into your document as you expect, check this first.

Laying the Foundation

The Document Generator will connect to a SQL Server database to download published Word templates to the user's hard drive. In addition, the Document Generator will need an Outlook Contact folder containing filled items representing client records. Before building the add-in, you will need to follow the steps in this section to create each of these items. For you list builders, the steps are:

1. Insert a new table into the Bravo Corp Database
2. Create the required Word Template files
3. Create a clients folder by importing the Northwind Customers table

The BravoCorp Database

For the code in this application to execute without a hitch, a new table named tblDocuments needs to be added to the Bravo Corp database. Create a new table using the schema specified in Table 3-1.

Table 3-1. The table schema for tblDocuments.

Column Name	Data Type	Length	Allow Nulls
DocumentID	int	4	no
DocumentName	varchar	255	no
DocumentDesc	varchar	255	yes
DocumentBinary	image	--	no

Before saving the table you need to set the following properties of the DocumentID column:

*Identity=yes

*Identity Seed=1

*Identity Increment=1

Your table should look similar to Figure 3-3. Save the table and open Word, you now need to create some document templates.

Word Templates

The templates used in the New Account scenario are:

* **General Services Agreement**: This document contains the contract details between Bravo Corp and a client.
* **Services Work Authorization:** This document contains authorization details for individual work orders from the client.

* **Status Report:** This document contains details relating to project status.

You can create these documents within Word or you can download them from the APress website. If you choose to build the documents yourself (how crafty of you! You must not be billable this morning), do not worry about having content applicable to the documents listed above. The reason for this is that it is the documents' bookmarks that really matter. If you feel like it, go ahead and make the documents as "authentic" as you like, just be sure and do as I say regarding those Bookmarks!

For each document you choose to create, insert the following bookmarks:

* CompanyName
* BusinessAddress
* BusinessAddressCity
* BusinessAddressState
* BusinessAddressPostalCode

To insert a bookmark, highlight some text within the document that will act as a placeholder (if you are currently staring at a blank document now is a good time to hammer your keyboard a couple of times). Select Insert➢Bookmark… from the main menu within Word. The Bookmarks dialog form is displayed (Figure 3-4). Insert a name for the selected text in the Bookmark Name field and press Add at the bottom of the form. Repeat this process until you have added all the remaining bookmarks.

Insert 0197f0302.tif

Figure 3-2. The Bookmarks Dialog Form.

Bravo Clients Outlook Data File

The Document Generator uses Outlook contact items as records containing client information. This is the information we will use at runtime to insert text into the document templates selected by the user. I recommend downloading the Bravo Corp Clients Outlook data file from the APress website. The data file can easily be recreated as it is an imported version of the Northwind Access database (and as we have discussed before, Northwind is included with Office and Visual Studio).

To import the Northwind database, perform the following steps:

1. Open Outlook and create a new Outlook data file by selecting File➢New➢Outlook Data File from the menu.
2. Select Office Outlook Personal Folders File from the New Outlook Data File window. Press OK.
3. Select the desired folder and name the file "Bravo Clients". Press OK.

4 In the Create Microsoft Personal Folders window, enter "Bravo Clients" in the Name field. Leave all other fields at their default values. Press OK.

5 Right-click on the newly created Bravo Corp folder in the Navigation Pane and select the New Folder option.

6 In the Create New Folder window, enter "Clients" in the name field. Select Contact Items from the Folder Contains drop down box. Press OK.

7 From the main menu, select File➢Import and Export.

8 The Import Export Wizard window is displayed. Select Import from another program or file from. Press Next.

9 Select Microsoft Access from the Select file type to import from list.

10 In the Import a File window, browse to the location of the Northwind.mdb file on your system. Press Next.

11 Choose the Clients folder created in step 4. Press Next.

12 Select "Import Customers into folders:Clients" and press the Map Custom Fields button.

13 Using the Map Custom Fields form (Figure 3-3), drag and drop items from the From area to the To area as listed in Table 3-2. Press OK.

Insert 0197f0303.tif

Figure 3-3. Map Custom Fields Form

Table 3-2. Import Field Mappings

Northwind Field Name	Contact Field Name
CompanyName	Company
Address	Business Street
City	Business City
Region	Business State
Postal Code	Business Postal Code
Fax	Business Fax
Phone	Business Phone

If the import was successful, the contents of the Clients folder will resemble Figure 3-4. Now that you have some sample client records for the add-in to work with, I can begin to show you how to build the Document Generator.

Insert 0197f0304.tif

Figure 3-4. The New Bravo Corp Clients Folder

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Creating the Add-In

The Document Generator add-in is self sufficient in terms of code. Unlike the Presentation Generator, this add-in does not rely on any code contained in other components. The following objects comprise the add-in:

* **appOutlook**: This class contains all the Outlook functions required by the add-in. This includes responding to Outlook events as well as setting up the add-in's Outlook interface (i.e. CommandBar buttons).
* **UserSettings**: This class contains the methods for storing and retrieving the user settings of the add-in.
* **Settings:** This form allows the user to change settings and calls the UserSettings methods for retrieving and saving the add-in's settings.
* **Documents:** This form lists available document templates available for creation and kicks off any additional processing. This is the main form of the add-in.
* **DocumentProcessor:** A form that allows the user to insert bookmark values for a specified Word document.
* **GetUpdates:** A form that retrieves all published presentations from a SQL Server and stores them on the user's file system.
* **Connect:** The class implementing the IDTExtensibility2 interface.

The appOutlook Class

Since Microsoft Outlook is the host application for the Document Generator, we will make great use of the objects and methods available to us in Outlook. The main purpose of the appOutlook class is to create the Bravo Tool CommandBar menu and setup the necessary Event Sinks. In addition, this class provides other utility methods available to the other classes in the project.

With Visual Studio.Net open, create a new Shared Add-In project. Add a class and name it "appOutlook".

NOTE: See Chapter 2 for detailed steps on how to create this type of project.

Imports Directive Section

The appOutlook class encapsulates all methods that access Outlook objects or call Outlook methods. This is a shared class to ensure that the methods and data of the class are shared by all other objects while the add-in is loaded. In addition to Outlook, the class will work with Office CommandBar objects and Windows Forms. Insert the following lines of code into the Imports section of the class:

```
Imports OL = Microsoft.Office.Interop.Outlook
Imports Microsoft.Office.Core
Imports System.Windows.Forms
```

The Outlook namespace is prefixed with "OL" to prevent naming conflicts between the Outlook and Office namespaces. Assigning a prefix like this is a good practice anytime there is a chance for naming conflicts among the different namespace objects.

The Declarations Section

The appOutlook class contains six class-level variables as follows:

```
Private Shared appOL As OL.Application
Private Shared fldClients As OL.MAPIFolder
Private Shared cbbBravoMenu As CommandBarPopup
Private Shared WithEvents cbbCreateDocs As CommandBarButton
Private Shared WithEvents cbbSettings As CommandBarButton
Private Shared WithEvents cbbGetUpdates As CommandBarButton
```

The appOL variable is a reference to the Outlook application and is used by the class's methods to call Outlook objects and functions. The fldClients variable contains a reference to the Outlook folder containing the Bravo Corp client records. The cbbBravoMenu variable is the Office CommandBar button that serves as the Bravo Tools menu. Both cbbCreateDocs and cbbSettings are Office CommandBars with event sinks that will allow the add-in to respond to their Click event and execute the code we attach later in this section.

The Class Methods

The appOutlook class is a shared class containing several methods to handle the Outlook "plumbing" for the other objects in the add-in. The class is setup as a shared class to make it easy to use. Since it is shared, there is no need to instantiate it prior to calling its methods. I like to avoid typing as much as possible and this strategy helps achieve this goal.

Okay, let's get going and actual create each method of the appOutlook class.

Setup

Prior to being used in the add-in, appOutlook needs to setup its environment. This is achieved by creating references in memory to the host Outlook application (appOL) and the location within Outlook of the Clients folder (fldClients), and by calling the SetupBravoMenu routine that will create the Bravo Tool menu.

The procedure looks like this:

```
Friend Shared Sub Setup(ByVal oApp As OL.Application, _
 ByVal EntryID As String, ByVal StoreID As String)

 appOL = oApp
 fldClients = GetFolder(EntryID, StoreID)
 SetupBravoMenu()
End Sub
```

This procedure takes the passed Outlook Application object and stores it for use within the class. The Connect class also has a reference to the Outlook host and we could have changed that reference so that it was available to appOutlook. The problem with that strategy is that it would break the principle of encapsulation. Now that Visual Basic.Net is fully object oriented, I am trying to break my old Visual Basic 6 habits and be a good boy.

Moving on, the GetFolder method is called to set a reference to the clients folder. Don't worry about the EntyrID and StoreID parameters for now. I will tell you all you need to know in moment. Last thing here is the call to the SetupBravoMenu function which adds the Bravo Tools menu to the Outlook application's main menu.

Shutdown

Since we have a Setup method to handle the class's initialization, let's go ahead and create the related Shutdown method before we (and when I say "we" I really mean "you") move on to the remaining functions. The Shutdown method could just as easily be called "JanitorialServices" because its sole purpose is to get out the broom, close Outlook, and put up until we need her again. Insert the Shutdown method as follows:

```
Friend Shared Sub ShutDown()
  appOL.Quit()
End Sub
```

Now, don't start thinking that nothin' much (some Texas lingo) is going on here. This little one liner does a lot of stuff, the most of important of which is the closing of any open Outlook windows and the logging off of the user session. Nice and tidy! Now that the foundation is set, it's time to build the rest of the class.

SetupBravoMenu

The SetupBravoMenu method handles the task of simply creating the add-in's menu. Sounds simple, yes? It is but as it turns out, this is the longest function of the class. Take in the code requirements before we step through each section

```
Private Shared Function SetupBravoMenu()
  Dim cbCommandBars As CommandBars
  Dim cbMenuBar As CommandBar

  ' Outlook has the CommandBars collection on the Explorer object.
  cbCommandBars = appOL.ActiveExplorer.CommandBars
  cbMenuBar = cbCommandBars.Item("Menu Bar")
  cbbBravoMenu = cbMenuBar.FindControl(tag:="Bravo Tools")
  If cbbBravoMenu Is Nothing Then
    cbbBravoMenu = cbMenuBar.Controls.Add( _
      Type:=MsoControlType.msoControlPopup, _
      Before:=6, Temporary:=True)
```

****

```
        With cbbBravoMenu
         .Caption = "&Bravo Tools"
         .Tag = "Bravo Tool"
         .TooltipText = "Bravo Crop Tools Menu"
         .OnAction = "!<DocumentGenerator.Connect>"
         .Visible = True
         'Create Documents Button
         cbbCreateDocs = .Controls.Add( _
           Type:=MsoControlType.msoControlButton, _
           Temporary:=True)

        With cbbCreateDocs
          .Caption = "Create Project Documents..."
          .Style = MsoButtonStyle.msoButtonCaption
          .Tag = "Create a set of Project Documents to send to a client."
          .OnAction = "!<DocumentGenerator.Connect>"
          .Visible = True
        End With
        'Settings Button
        cbbSettings = .Controls.Add( _
          Type:=MsoControlType.msoControlButton, _
          Temporary:=True)
        With cbbSettings
          .Caption = "User Settings..."
          .Style = MsoButtonStyle.msoButtonCaption
          .Tag = "Change the Document Generator User Settings."
          .OnAction = "!<DocumentGenerator.Connect>"
          .Visible = True
        End With
        'Download Updates Button
        cbbGetUpdates = .Controls.Add( _
          Type:=MsoControlType.msoControlButton, _
          Temporary:=True)
         With cbbGetUpdates
          .Caption = "Dowload Templates..."
          .Style = MsoButtonStyle.msoButtonCaption
          .Tag = "Download additional Document Templates.."
          .OnAction = "!<DocumentGenerator.Connect>"
          .Visible = True
        End With
      End With
    End If

  End Function
```

The first section of the function handles the process of adding a button to Outlook's Menu Bar CommandBar. You can add the button to any of the menu bars in Outlook or you could even create a new CommandBar specifically for the add-in. I chose to the first strategy:

```
' Outlook has the CommandBars collection on the Explorer object.
  cbCommandBars = appOL.ActiveExplorer.CommandBars
  cbMenuBar = cbCommandBars.Item("Menu Bar")
  cbbBravoMenu = cbMenuBar.FindControl(tag:="Bravo Tools")
  If cbbBravoMenu Is Nothing Then
    cbbBravoMenu = cbMenuBar.Controls.Add( _
      Type:=MsoControlType.msoControlPopup, _
      Before:=6, Temporary:=True)
    …
  End if
```

> NOTE: I like to incorporate add-in menus as part of the host application's main menu as it conserves space and is less likely to confuse a user.

The method determines if the menu already exists and if it does not, the Bravo Tools menu is added to the Menu Bar and its properties are set.

Notice that the Bravo Tools CommandBar button's Temporary property is set to True. This means the button will be removed automatically when Outlook is shutdown. In addition, the Before property is set to "6". Setting the 'Bravo Tools' menu in position just after the default 'Tools' item (assuming no menu customizations by the user).

> TIP: If you are feeling frisky, edit the code to look for the location of the Tools menu and then place the Bravo Tools menu next to it.

> NOTE: There are advantages and disadvantages to setting the Temporary parameter to either True or False. The advantage of a temporary CommandBarButton is that there is no need to write code to remove the add-in's buttons upon disconnection with the host application. The disadvantage is that the CommandBar must be created each time the application loads the add-in. In addition, if the user moves the CommandBar the new location will not be saved between application sessions.

Once created, the newly added menu button's Caption, Tag, ToolTipText, and OnAction properties are set.

```
With cbbBravoMenu
  .Caption = "&Bravo Tools"
  .Tag = "Bravo Tool"
  .TooltipText = "Bravo Crop Tools Menu"
  .OnAction = "!<DocumentGenerator.Connect>"
  .Visible = True
…
End With
```

If you remember from Chapter 2, the OnAction property is set to the add-in project's Connect class. This tells the command button to run the Document Generator add-in each time the button is clicked. Okay, we now have a menu control, let's add some buttons to it.

> TIP: The OnAction property needs to contain the Program ID of the add-in and the name of its class containing the IDTExtensibility2 interface (the Connect class in this case). Instead of using the project name here, you could use reflection to discover the add-in's name and insert it as the value for the OnAction property.

The remainder of the procedure adds the two CommandBarButton objects, cbbCreateDocs and cbbSettings, to cbbBravoMenu and set their properties. Like the Bravo Menu object, both of these buttons are set as temporary CommandBar buttons.

```
 'Settings Button
cbbSettings = .Controls.Add( _
  Type:=MsoControlType.msoControlButton, _
  Temporary:=True)
With cbbSettings
  .Caption = "User Settings..."
  .Style = MsoButtonStyle.msoButtonCaption
  .Tag = "Change the Document Generator User Settings."
  .OnAction = "!<DocumentGenerator.Connect>"
  .Visible = True
End With
'Download Updates Button
cbbGetUpdates = .Controls.Add( _
  Type:=MsoControlType.msoControlButton, _
  Temporary:=True)
 With cbbGetUpdates
  .Caption = "Dowload Templates..."
  .Style = MsoButtonStyle.msoButtonCaption
  .Tag = "Download additional Document Templates.."
  .OnAction = "!<DocumentGenerator.Connect>"
  .Visible = True
  End With
```

PickContactsFolder

The DocumentGenerator relies on the existence of an Outlook Contacts folder for storing client records. Since we want to be nice and flexible, we will build some logic that allows the user to pick any folder in the Outlook folder hierarchy whose default item type is ContactItem.

Ladies and gentlemen, the PickContactsFolder method:

```
Friend Shared Function PickContactsFolder() As String()
```

```
    Dim nsMAPI As OL.NameSpace
    Dim fld As OL.MAPIFolder
    Dim str(2) As String
    nsMAPI = appOL.Application.GetNamespace("MAPI")

    PickFolder:
     fld = nsMAPI.PickFolder

     If fld Is Nothing Then
       Exit Function
     Else
      If fld.DefaultItemType <> OL.OlItemType.olContactItem Then
        MsgBox("Please pick a folder containing Contact Items.")
        GoTo PickFolder
      Else
        str(0) = fld.FolderPath
        str(1) = fld.EntryID
        str(2) = fld.StoreID
        Return str
      End If
     End If
  End Function
```

Using the MAPI namespace (currently the only namespace available within Outlook), the PickFolder method executes to display a dialog box (Figure 3-4).

```
    Dim nsMAPI As OL.NameSpace
    Dim fld As OL.MAPIFolder
    Dim str(2) As String
    nsMAPI = appOL.Application.GetNamespace("MAPI")

    PickFolder:
     fld = nsMAPI.PickFolder
```

This dialog form contains a TreeView of all Outlook folders and makes things nice and simple for a user to select an Outlook folder. The code pauses until the user selects a folder and closes the form (or, they just cancel). PickFolder then returns a string array containing the important information needed to retrieve the selected folder at any time.

Insert 0197f0305.tif

Figure 3-5. The Pick Folder Dialog Box

If a folder was not selected, fld equates to nothing and the function is exited However, if a folder is selected, the folder type is checked to ensure it contains ContactItems before returning the folder as the value of the method. If the folder does not contain ContactItems, the user is notified and the PickFolder dialog box is displayed again.

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

```
If fld Is Nothing Then
  Exit Function
 Else
  If fld.DefaultItemType <> OL.OlItemType.olContactItem Then
   MsgBox("Please pick a folder containing Contact Items.")
   GoTo PickFolder
  Else
   str(0) = fld.FolderPath
   str(1) = fld.EntryID
   str(2) = fld.StoreID
   Return str
  End If
 End If
```

Once a contacts folder has been selected certain property values we care about are returned within a string array. The string array contains three little nuggets of data pertaining to the selected folder.

str(0) = fld.FolderPath

Our full Outlook folder path replete with user-friendly value for display (if I were to select my default Contacts folder as my clients folder. In this instance, the FolderPath property would be \\Mailbox - Ty Anderson\Contacts

str(1) = fld.EntryID

The unique identifier of the selected folder. Anytime an object is first saved within Outlook, the MAPI data store assigns the EntryID (supplied by Outlook on creation)

str(2) = fld.StoreID

The unique identifier of the data store containing the EntryID.

> CAUTION: Be careful when using the EntryID and StoreID of an Outlook Item. If an item is moved from one folder to another, both properties will change as the MAPI store will generate new values. This could cause some major problems if you do not allow for these changes. For example, in this add-in, if the user were to change the location of the clients folder after selecting it in the Settings form, the add-in would not be able to access it. In order for the add-in to again function properly, the user would need to update the location of the Bravo Clients folder in the Settings form.

One last bit here is the fact that I have used an old school, VB6ish, label named PickFolder. I did this because I wanted to force the user to select a folder whose default type is ContactItems. If the user does not select the desired folder type, using the label, I just send them back to the beginning of the function and have them start over. This could have been done with a recursive function but isn't. Sometimes you do what works versus what is the prettiest.

Soooo…using this function the user can now specify which folder contains their client records. Now what? Now entering Stage Right…ListContacts!

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

ListContacts

What good is it to know which folder contains the client contact records if all that will happen is they just sit and wait for someone to love them? To put the items residing in the specified contacts folders to good use, let's add a helper function named ListContacts. This handles all the details of rummaging through the client records and adding them to a list in a ComboBox. All this sweet little function asks is that you pass it a reference to the ComboBox you want filled. Let's take a detailed look.

The ListContacts method accepts a ComboBox type variable and fills it with the names of client contacts:

```
Friend Shared Function ListContacts(ByVal ctl As ComboBox)
  Dim fld As OL.MAPIFolder
  Dim itms As OL.Items
  Dim itm As OL.ContactItem
  Dim strName As String
  Dim i As Integer

  fld = GetContactsFolder()
  itms = fld.Items
  itms.Sort("[LastName]", False)

  For i = 1 To itms.Count
    itm = itms(i)
    strName = itm.LastName & "," & itm.FirstName
    ctl.Items.Add(strName)
  Next i

End Function
```

ListContacts begins by calling the GetContactFolder function and assigning the return to value to MAPIFolder variable object. This object's items are then assigned to an Outlook Items variable used for looping through the contents of the folder. Before starting the loop, the contents of the folder are sorted by LastName in order to present the data in a manner expected by the user. The method finishes by looping through each item and adding items to the passed ComboBox control. The items are listed in LastName, FirstName format.

Okay, we can specify which folder contains the client contact records. We can also fill a ComboBox full of these client records. What else do we need? What if we provided a way to retrieve the actual ContactItem once selected?

GetContact

GetContact is a simple function that will return any Outlook ContactItem you ask it to (assuming it exists and GetContact doesn't get distracted while looking for it). The

GetContact method accepts a single string (the name of a client) as a parameter and returns an Outlook ContactItem:

```
Friend Shared Function GetContact(ByVal ContactName As String) _
 As OL.ContactItem

 Dim itms As OL.Items
 Dim itm As OL.ContactItem

 itms = fldClients.Items
 itm = itms(ContactName)

 Return itm

End Function
```

Using the class's reference to the Clients folder, the method uses the folder's items collection and the ContactName variable is used to retrieve the related ContactItem. Once found, it is returned as the value of GetContactFolder.

> Caution: When searching for a ContactItem, be sure and use a name in the *FirstName LastName* format. This can be problem given that ListContacts fills a ComboBox in the *LastName, FirstName* format. If the code above used this second format, the desired contact would not be found.

What happens if you give a pig a pancake? She will most certainly want some syrup. What happens if you let the user select a client record? They will probably want to bug 'em with an email! I am a problem solver and this is one problem I can certainly solve. I need your help though to type the code. Let's see how this is done.

CreateEmail

One of the great things about Outlook is that it can be used to send email. I know it's incredible but as it turns out this is actually the number one reason people open this little application in the first place!

Don't let my snide comments detract from the fact that sending email in Outlook is as easy as falling out of bed (how is falling out of bed? Just ask my two year old). Now, let's take a looksy.

The CreateEmail method creates an Outlook MailItem and then inserts values into a few of the MailItem's relevant properties. The end result is a new email filled with the selected contact's email address. In addition, CreateEmail goes ahead and attaches the document(s) created earlier in the workflow. Whoa! Now don't get all obsessive compulsive on me. We have not written the code for the document creation yet…so don't panic, you haven't missed anything!

```
Friend Shared Sub CreateEmail(ByVal FileNames() As String, _
 ByVal ClientName As String)
```

```
    Dim mi As OL.MailItem
    Dim ci As OL.ContactItem
    Dim i As Integer
    Dim TotalCount As Integer

    TotalCount = FileNames.GetUpperBound(0)
    ci = GetContact(ClientName)
    mi = appOL.CreateItem(OL.OlItemType.olMailItem)
    mi.To = ci.Email1Address
    mi.Subject = "Event Documents for " & ci.CompanyName

    For i = 0 To TotalCount - 1
      mi.Attachments.Add(FileNames(i))
    Next

    mi.Display()

  End Sub
```

The method begins by determining the size of the passed string array for use in the loop that completes the procedure.

```
    TotalCount = FileNames.GetUpperBound(0)
```

We need to know how many documents, later in the procedure, when we add attachments to the email.

Next, GetContact is called to retrieve the desired ContactItem. It then uses the class's Outlook application object to create a new MailItem and set its properties.

```
    ci = GetContact(ClientName)
    mi = appOL.CreateItem(OL.OlItemType.olMailItem)
    mi.To = ci.Email1Address
    mi.Subject = "Event Documents for " & ci.CompanyName
```

The email's To field is set to the ContactItem's email address while the Subject field makes use of the company name in the contact record.

Before displaying the MailItem, the procedure loops through the filenames contained in the passed string array. Each string in the array is used to add a new attachment to the email.

```
    For i = 0 To TotalCount - 1
      mi.Attachments.Add(FileNames(i))
    Next
```

To finish things off, the MailItem is displayed (Figure 3-6) and the user can do with it as the wish – add a nice little note, add more recipients, send it or cancel it.

```
    mi.Display()
```

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Insert 0197f0306.tif

Figure 3-6. The Completed Email with Documents Attached

So are you feeling good about yourself? We (and when I say we I mean you) have managed to create several useful methods for our class and it isn't even lunchtime yet. A most productive start to the day, don't you think?

We have one more "thing" to accomplish. If you remember, we declared three class level variables at the beginning of this class name cbbCreateDocs, cbbSettings, and cbbGetUpdates using the With Events keyword. This just means we can respond to any events these variables raise.

Since each of these three variables is a CommandBar object, their click event really needs SOMETHING to do. Don't worry, I already have some ideas on how to put them to good use so save your thinking for what you would rather have for lunch, Taco Bell or Wendy's?

Let's take them one a time, shall we? Okay from the top…

cbbCreateDocs_Click

This little button starts the whole document generation process. It is, effectively, the most popular button of the Document Generator add-in. This status is overrated really because all the only heavy lifting here is the appearance of the Documents form we will create in just a bit.

The Click event for the cbbCreateDocs CommandBar looks like this:

```
Private Shared Sub cbbCreateDocs_Click(ByVal Ctrl As _
 Microsoft.Office.Core.CommandBarButton, ByRef CancelDefault As Boolean) _
 Handles cbbCreateDocs.Click

 Dim frmDocuments As New Documents
 frmDocuments.Show()

End Sub
```

Easy enough? We just declare a variable of type Documents and then we call its Show method.

cbbSettings_Click

This nifty button is also very popular with the users. By clicking this button, the user will be shown the Settings form which allows them to customize the settings for the add-in.

The Click event for the cbbSettings is now presented for your entertainment:

```
Private Shared Sub cbbSettings_Click(ByVal Ctrl _
 As Microsoft.Office.Core.CommandBarButton, ByRef CancelDefault As Boolean) _
 Handles cbbSettings.Click
```

```
    Dim frmSettings As New Settings
    frmSettings.Show()

End Sub
```

Two down, one to go...I am starting to smell lunch.

cbbGetUpdates_Click

GetUpdates is just like the last two "subs" except that it displays the GetUpdates form. Type some code until the Click event for cbbGetUpdates resembles the following block:

```
Private Shared Sub cbbGetUpdates_Click(ByVal Ctrl _
  As Microsoft.Office.Core.CommandBarButton, _
  ByRef CancelDefault As Boolean) Handles cbbGetUpdates.Click

  Dim frm As New GetUpdates
  frm.ShowDialog()
End Sub
```

You have just completed the appOutlook class, not a bad bit of work. Get ready though, because we are going to go full boar in the next sections where we write the guts of the Document Generator. But before that, I am going to step out for some tacos.

The Documents Form Class

We need a form to act as the primary user interface of the add-in. This form needs to be simple but powerful. By simple, I mean that the form does not need to confuse our friendly little user (remember, we want them to like our application and use it – it keeps us employed). By powerful I mean that it needs to perform some highly productive tasks like:

* retrieve a listing of available document templates
* allow the user to select templates for creation
* generate the selected documents

Here is how the Documents form does its job. First, the user to selects their desired documents from a list of available document templates. These templates reside in a folder specified by the user in the Settings form (discussed later). Second, a client record is selected from a ComboBox filled with the names of all Bravo Corp Clients (this is our little appOutlook class already at work!). Third, these selections are passed to the DocumentProcessor form (we will create this shortly) for document creation.

Now you know what the Documents form does, so let's move on and create it. The form contains:

three Label controls: lblClients, lblAvailableDocs,lblSelectedDocs,

two ListBox controls: lstAvailableDocs,lstSelectedDocs,

aComboBox control cboClients

six Button controls cbSelectOne, cbRemoveOne, cbSelectAll, cbRemoveAll, cbOk, and cbCancel.

Add a new form to the project, name it "Documents" and then layout the controls as illustrated in Figure 3-5.

Insert 0197f0307.tif

Figure 3-7. The Documents Form Control Layout

When the form loads a call is made

Imports Directives

The Documents form will make use of the Windows File system so edit the Imports section so that it resembles this:

```
Imports System.IO
Imports System.Windows.Forms
```

The System.IO namespace provides everything we need for opening, reading, and closing files stored on the user's system. The System.Windows.Forms namespace handles all details of the creating Windows forms and is automatically added when you create a new form. Now that that's all set, let's write some code.

Methods

As I mentioned just a moment ago, the Documents form has three main objectives – list available templates, allow the user to select their documents they wish to create, and create the selected documents. We achieve through the use of four custom functions and by responding to eight control or form events. The custom methods are called in the events so we will begin by writing them first.

GetFiles

By now you are well aware that the Documents form needs to retrieve a list of available document templates on the user's machine. As luck would have it, this is exactly what I have in mind for a function we will call GetFiles. The GetFiles takes a look in the templates folder specified in the user settings. As it loops through the files in the folder, GetFiles fills the passed ListBox control with the names of each file Word Document it finds. Take a look:

```
Private Function GetFiles(ByVal ctl As ListBox)
  Dim diFolder As New DirectoryInfo(UserSettings.TemplatesFolder)
  Dim fi As FileInfo

  For Each fi In diFolder.GetFiles("*.dot")
    ctl.Items.Add(fi.Name)
  Next
```

```
End Function
```

There really isn't anything all that special here. We just declare a couple of variables, assign values to them, and perform a loop. The object of special interest here is the FileInfo typed variable name fi. Using this type of object, just about anything you want related to a file is at your fingertips.

We kill two birds with one stone by declaring the diFolder variable as a DirectoryInfo class and assigning it the path of the user's templates folder. The DirectoryInfo object will return a collection of files of any type you specify by calling its GetFiles method (no relation to our Documents.GetFiles method. It has a totally different family and a much more sordid history.) and telling which file extensions to retrieve. Using the returned collection of files, our little GetFiles custom method loops through all the goodies in the templates folder and adds the name of each to the ListBox that was handed to it (via its ctl argument). The end result is a lovely little ListBox filled with Word template file names.

> Tip: Notice, I didn't alphabetize the files before adding them to the ListBox. This is in large part because I am lazy but it is also because I wanted to give you something to do.

> Note: The .Net Framework provides several objects for manipulating files and directories and there is a lot of overlap. Next time you are having trouble sleeping at night, open up the .Net documentation and learn something new.

Okay, so we have a ListBox filled with file names. and we need to move them from one control to another. Hold your excitement and keep reading.

MoveSelected

Our user is bound to get excited at the sight of a ListBox filled with available document templates. So excited, in fact, that they may just start using their mouse to make selections uncontrollably. In the end, they are sure to calm down and choose only the documents the really need so we better be ready.

The MoveSelected method moves the selected item from one ListBox to another. All it needs to know is which control is the source and which is the destination.

```
Private Function MoveSelected(ByVal SourceList As ListBox, _
 ByVal TargetList As ListBox) As Boolean
 Try
  TargetList.Items.Add(SourceList.SelectedItem)
  SourceList.Items.Remove(SourceList.SelectedItem)
  Return True
 Catch ex As Exception
  Return False
 End Try
End Function
```

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Using the Items collection of each ListBox, we either add the SelectedItem form the SourceList or remove it. It all depends on if the ListBox is the target or the source. MoveSelected returns a Boolean value based on whether or not the operation was successful.

Personally, I like the simplicity of this function. Not only is it simple, it is primed for reuse in another project because of its generic nature. As you may have already realized, this makes me feel like I have written some code that even the most stolid programmer would appreciate.

MoveAll

One thing I try to do when developing applications is anticipate the gripes, I mean, the needs of my users. The last thing I really want to hear is how my application doesn't do what it should. This kind of user attitude can really ruin a good cup of coffee. Therefore, in an attempt to reduce user frustration, make them even more productive, and allow us all to enjoy our cup of Joe, let's add another function to move all items in a ListBox to another ListBox.

The MoveAll method moves all items, with no regard for items currently selected, from one ListBox control to another. It is the same as the MoveSelected function but different. See for yourself:

```
Private Function MoveAll(ByVal SourceList As ListBox, _
 ByVal TargetList As ListBox) As Boolean

 Try
  Dim si

  For Each si In SourceList.Items
   TargetList.Items.Add(si)

  Next
  SourceList.Items.Clear()
  Return True

 Catch ex As Exception
  Return False
 End Try

End Function
```

Just like MoveSelected, this function accepts a source and destination ListBox as parameters. Using a generic object (si), a loop is implemented to add new items to the TargetList by cycling through each item in the SourceList's Items collection. Once each and every item from the SourceList has been successfully added to the target control, the SourceList Item's collection is cleared of all items.

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

> Tip: Because you are incredibly smart and notice such things…I know you noticed that when I declared the si variable in the previous chunk of code, I did not assign a type. This is because, to be brutally honest, I am unaware of an Item type for a ListBox control. But since VB.Net is fully object oriented, I just declared si as a generic object and moved on. Notice how it works just fine (but your purists out there are sure to inform of the performance hit, blah!).

Like I said, MoveAll is just like the previous function but just a tad different because it moves EVERYTHING! Our users will feel very efficient. This feeling of efficiency should lead to a feeling of being more productive. And because they feel so gosh darn productive, they will be less likely to send some incredibly rude email our way. This makes me want some more coffee.

GenerateDocs

Let's take a quick review of our user scenario (or workflow, whatever) up to this point. The user needs to send their client some new documents relevant to their event. In this scenario, the Account Manager has opened the Documents form by clicking the cbbCreateDocs button residing in the Bravo Tools menu. Once the Documents form was open the Account Manager picked at least one available document template for creation. These selections are now sitting the lstSelectedDocs control. In addition, the user (still the Account Manger here) has used the cboClients combo box to specify which client will receive these documents. That brings pretty much brings us up to date.

Insert NEW IMAGE HERE!!

The GenerateDocs method handles the task of running the DocumentProcessor class against each selected document template in the lstSelectedDocs control. I will attempt to run through this fairly quickly so we can take a look at the DocumentProcessor, but stay with me if you can because this stuff is important.

Take a minute to read through GenerateDocs before I explain everything it does.

```
Private Sub GenerateDocs()

    Dim i As Integer
    Dim strClient As String
    Dim strFile As String
    Dim strFiles(lstSelectedDocs.Items.Count) As String

    strClient = cboClients.SelectedItem
    Try
      For i = 0 To lstSelectedDocs.Items.Count - 1
        Dim dp As New DocumentProcessor
        strFile = lstSelectedDocs.Items(i)
```

```
        With dp
          .Client = appOutlook.GetContact(strClient)
          .FileName = strFile.ToString
          .OpenPath = UserSettings.TemplatesFolder
          .SavePath = UserSettings.SaveFolder
          .OpenDoc()
          .PopulateForm()
          .ShowDialog()
          'Populate String Array.  These will be inserted as
          'Attachements to an email.
          If .DocSaved Then
            strFiles(i) = .SavePath & "\" & .SavedFileName
          End If
          .Visible = False
          .Dispose()

        End With

      Next
    Catch ex As Exception
      MsgBox(ex.Message)
    End Try

    appOutlook.CreateEmail(strFiles, strClient)
  End Sub
```

At first glance it looks like a lot, and it is. However, it really is quite simple. We start by first declaring some worker variables, the most notable being a String array. Notice the use of the source control's Items.Count property to set the array's size:

```
    Dim i As Integer
    Dim strClient As String
    Dim strFile As String
    Dim strFiles(lstSelectedDocs.Items.Count) As String
```

This sets the array's size to the same number of items that have been selected and helps avoid the need to use ReDim later. Next, we store the value of the client name selected from cboClients

```
    strClient = cboClients.SelectedItem
```

and start a loop to process each selection.

Within the loop, each selected item is handed off to an instance of the DocumentProcessor class which handles the actual document creation details. A String is set to the value of the current item referenced in the loop.

```
    For i = 0 To lstSelectedDocs.Items.Count - 1
      Dim dp As New DocumentProcessor
      strFile = lstSelectedDocs.Items(i)
```

```
    …
Next
```

Next, we setup the DocumentProcessor form prior to displaying it.

```
With dp
.Client = appOutlook.GetContact(strClient)
.FileName = strFile.ToString
.OpenPath = UserSettings.TemplatesFolder
.SavePath = UserSettings.SaveFolder
.OpenDoc()
.PopulateForm()
.ShowDialog()
…
End With
```

We set the Client property to an Outlook ContactItem representing the selected client. The OpenPath and SavePath properties are set to the user defined add-in values.

Next, the document is opened and a call is made to PopulateForm. This method scans the open Word document for any existing Bookmarks. If it finds any, it creates a series of text boxes and labels on the DocumentProcessor form – each representing a single bookmark. Finally, the DocumentProcessor form opens as a dialog box and the function waits for the user to do their thing and close the form.

Upon completing the DocumentProcessor (or canceling out), GenerateDocs first determines if the document was saved by checking the DocSaved property of the DocumentProcessor.

If the document has been saved then the path and file name are added to strFiles.

```
If .DocSaved Then
  strFiles(i) = .SavePath & "\" & .SavedFileName
End If
```

Finally the DocumentProcessor form is hidden and disposed of.

```
.Visible = False
.Dispose()
```

Wrapping things up, the appOutlook class's CreateEmail method is passed the strFiles array to create and display a new email addressed to the client with the newly created documents attached.

```
appOutlook.CreateEmail(strFiles, strClient)
```

One again, we have all the functions needed to make things happen. The next step is to hook our functions up with the appropriate form and/or control events needed to put our code in motion.

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Form and Control Events

Because we have done such a good job with our functions, the event code for the Documents form and its controls are super simple and will only take a your left hand to type (you lefties out only need your right hand).

Let's take them in logical order, starting with the loading of the Documents form.

Documents_Load

The Documents_Load event prepares the form by filling the lstAvailableDocs and cboClients controls with data as follows:

```
Private Sub Documents_Load(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles MyBase.Load

 GetFiles(lstAvailableDocs)
 appOutlook.ListContacts(cboClients)

End Sub
```

cboClients_Click

The cboClients_Click event adds a little user friendly interaction by reversing the current state of the ComboBox control's DroppedDown property:

```
Private Sub cboClients_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cboClients.Click

 cboClients.DroppedDown = Not cboClients.DroppedDown

End Sub
```

This method allows the user to simply click on the control to view or hide the control's contents.

cbSelectOne_Click

The cbSelectOne_Click event calls the MoveSelected method to move the selected item from lstAvailableDocs to lstSelectedDocs:

```
Private Sub cbSelectOne_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cbSelectOne.Click

 MoveSelected(lstAvailableDocs, lstSelectedDocs)

End Sub
```

cbRemoveOne_Click

The cbRemoveOne_Click event calls the MoveSelected method to move the selected item from lstSelectedDocs to lstAvailableDocs:

```
Private Sub cbRemoveOne_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cbRemoveOne.Click

  MoveSelected(lstSelectedDocs, lstAvailableDocs)

End Sub
```

cbSelectAll_Click

The cbSelectAll_Click event calls the MoveAll method to move the selected item from lstAvailableDocs to lstSelectedDocs:

```
Private Sub cbSelectAll_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cbSelectAll.Click

  MoveAll(lstAvailableDocs, lstSelectedDocs)

End Sub
```

cbRemoveAll_Click

The cbRemoveAll_Click event calls the MoveAll method to move the selected item from lstSelectedDocs to lstAvailableDocs:

```
Private Sub cbRemoveAll_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cbRemoveAll.Click

  MoveAll(lstSelectedDocs, lstAvailableDocs)

End Sub
```

cbOK_Click

The cbOK_Click event calls the GenerateDocs method and, upon its completion, closes the Documents form:

```
Private Sub cbOK_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cbOK.Click

  GenerateDocs()
  Me.Close()
```

```
End Sub
```

cbCancel_Click

The cbCancel_Click event closes the form without any further processing:

```
Private Sub cbCancel_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cbCancel.Click


 Me.Close()


End Sub
```

And so, just like that, we have completed the Documents form. But don't get too excited – we still have lots to do. Next on the agenda is the DoumentProcessor form. which handles all aspects of creating each document and as now know, is used extensively by the Documents form.

The DocumentProcessor Form Class

As you saw in the previous section, the DocumentProcessor form is called upon by the Documents form to do the job of creating a new document and filling it with details from a specified client record. This task is performed through the execution of the following tasks:

1. Opening the Word Document template
2. Reading the document's Bookmarks collection
3. Populating the DocumentProcessor with controls representing each Bookmark
4. Inserting the values from the form's controls into their corresponding Bookmark within the document.
5. Saving the document to the UserSettings.SaveFolder path.

The form is completely subordinate to the Documents form and is not accessed from anywhere else within the project. In addition, the DocumentProcessor only handles one document at a time.

At design-time, the form contains one Panel control (pnlTitle), one Label control (lblTitle), and two Button controls (cbOK and cbCancel). More controls will be added at run-time which will allow the user to enter values for the document's Bookmarks.

Add a new form to the project and name it "DocumentProcessor". Using Figure 3-6 as a guide, layout the controls in a similar manner.

Insert 0197f0308.tif

Figure 3-8. The DocumentProcessor Form Control Layout

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Imports Directives

This form will make use of Microsoft Word and Outlook. Edit the Imports section of the form class to look just like this:

```
Imports W = Microsoft.Office.Interop.Word
Imports OL = Microsoft.Office.Interop.Outlook
Imports Frm = System.Windows.Forms
```

This puts Word and Outlook at our mighty little fingertips. In addition, since some of the objects in the namespaces may conflict, I added a prefix to each statement so we can qualify the objects later on.

Variable Declaration

We are going to need several variables for the different tasks the DocumentProcessor will perform. For now, just go ahead and insert the following lines of code into the class.

```
Private m_appW As W.Application
Private m_Doc As W.Document
Private m_OpenPath As String
Private m_SavePath As String
Private m_FileName As String
Private m_SavedPathAndFileName As String
Private m_Contact As OL.ContactItem
Private m_Documents As System.Windows.Forms.ListBox
Private m_WordIsRunning As Boolean
Private m_DocSaved As Boolean
```

Since all of these are declared as Private class variables, we need to add a property procedure for each one that we want to expose to other classes in the project. We could have just declared them all as Public fields but we have the time to be purists about it and do this correctly (or at least I do). Besides, some of the variables need to be defined as read only properties of the class and we need a correct property procedure to accomplish this.

Properties

Unlike the other classes we have created for this project, this class already has several properties (and methods now that you mention it). In addition to the properties available to any class inherited from the System.Windows.Forms.Form class, the DocumentProcessor needs the following custom properties.

> Note: The fact that there are already properties and methods for the form are not a big deal at all. A form is just a class with a visual aspect to it. I just don't want you to freak out later when hit that "." and Intellisense lists a couple dozen or more properties than the custom properties we are about to create.

GetBookmarks

GetBookmarks is a read-only property that returns the Bookmarks collection of the m_Doc variable.

```
Public ReadOnly Property GetBookmarks() As W.Bookmarks
  Get
    If Not m_Doc Is Nothing Then
      Return m_Doc.Bookmarks()
    End If
  End Get
End Property
```

DocSaved

The DocSaved property is a read-only property and returns the value of the m_DocSaved class variable. This property is set to True upon the successful completion of the SaveDoc method (discussed in the next section covering the class's methods):

```
Public ReadOnly Property DocSaved() As Boolean
  Get
    Return m_DocSaved
  End Get
End Property
```

The property allows any object using the DocumentProcessor class to determine if the document was saved and to respond appropriately.

FileName

The FileName property is a read/write property representing the name of the file to be processed:

```
Public Property FileName() As String
  Get
    Return m_FileName
  End Get
  Set(ByVal Value As String)
    m_FileName = Value
  End Set
End Property
```

In our scenario, the file name one of the selected Word documents stored in the templates folder.

OpenPath

The OpenPath property is a read/write property representing the folder where the document defined in the FileName property resides:

```
Public Property OpenPath() As String
  Get
    Return m_OpenPath
  End Get
  Set(ByVal Value As String)
    m_OpenPath = Value
  End Set
End Property
```

SavePath

The SavePath property is a read/write property storing the folder location where the completed documented should be saved:

```
Public Property SavePath() As String
  Get
    Return (m_SavePath)
  End Get
  Set(ByVal Value As String)
    m_SavePath = Value
  End Set
End Property
```

SavedFileName

The SavedFileName property is a read-only property that stores the name of the file after it has been saved to the SavePath location:

```
Public ReadOnly Property SavedFileName() As String
  Get
    Return m_SavedPathAndFileName
  End Get
End Property
```

The m_SavedPathAndFileName variable is updated in the SaveDoc method (discussed in the next section).

Client

The Client property is a read/write property representing the Outlook ContactItem containing the client's contact information:

```
Public Property Client() As OL.ContactItem
  Get
    Client = m_Contact
  End Get
  Set(ByVal Value As OL.ContactItem)
    m_Contact = Value
```

```
    End Set
  End Property
```

As you can see, each property method exposes each of the class variables for access by other classes. Now that our variables/properties are done, we can focus our attention on the class methods.

Methods

Okay, this is where we get funky. That's right we have a perfectly good class except that it really doesn't do anything. Let's change that right here

The DocumentProcessor class has six custom methods, each handling a different aspect of creating a new Word document. In addition to these methods the New and Dispose methods are customized just a tad for our specific purposes. Let's take a look at New.

New

Before the form displays itself to the user, you will want to properly setup its environment. The New method is a great location for this code as it is the class's constructor method.

> Tip: A constructor method runs anytime a new instance of the class is instantiated. All this means to you and I is that the constructor is the perfect place for placing code that needs to run each time the class is created.

Edit the form's New method to include the custom logic for opening Microsoft Word. The new New method should look like this:

```
  Public Sub New()
    MyBase.New()

    'This call is required by the Windows Form Designer.
    InitializeComponent()

    'Add any initialization after the InitializeComponent() call
    Try
      m_appW = GetObject(, "Word.Application")
      m_WordIsRunning = True
    Catch ex As Exception
      If Err.Number = 429 Then
        m_appW = CreateObject("Word.Application")
        m_WordIsRunning = False
      Else
        Throw New System.Exception("Microsoft Word Automation error.")
      End If
    End Try
```

```
End Sub
```

To function properly, the DocumentProcessor really needs for Word to be running. Word does not have to be visible and the center of the user's attention. It just needs to be running as a process on the user's machine. The tricky part about this is we need to allow for the fact that the user just might have already opened Word and started updating their resume (at least that's what people do in my office). Before creating a new instance of Word, the method determines whether or not Word is currently running as follows:

```
'Add any initialization after the InitializeComponent() call
Try
 m_appW = GetObject(, "Word.Application")
 m_WordIsRunning = True
```

The class name of Word is passed to the GetObject function to check for an existence of Word running in the operating system. If one is found, a flag is set to remind us later this was the case. This little nugget of info will be useful later in the Dispose method.

> Note: Anytime you use GetObject or CreateObject, VB.Net creates a wrapper object to handle the calls to the COM object. This is known as an interop assembly. If an interop assembly already exists, it is used. If not, Visual Studio asks if you would like for it to create one.

If Word is not running the code will trip an error which the following Catch block handles quite nicely:

```
Catch ex As Exception
 If Err.Number = 429 Then
  m_appW = CreateObject("Word.Application")
  m_WordIsRunning = False
 Else
  Throw New System.Exception("Microsoft Word Automation error.")
 End If
```

Error number 429 ("Cannot Create ActiveX Component") occurs as a result of Word already running as a process. The error message could be a bit more useful but I will save that discussion next time I have the ear of a Microsoftie. Anyways, in this case error 429 means Word is not running and desperately needs to be opened. Fortunately, GetObject has a sister function named CreateObject to do just that. Once Word is successfully opened, a m_WordIsRunning flag is set to False specifying Word was started by the DocumentProcessor. If any other error occurred, a custom exception is thrown.

> Tip: How do I know the error number is 429? That's easy! I run the code through the debugger while commenting out any Try…Catch blocks that exist. I then use the Exception object to determine the error number and write code to respond as I see fit.

> Tip: If you can be assured that Word will not be opened when running the add-in, you could utilize the CreateObject method. It is guaranteed to open create a new instance of the targeted class each

and every time it is called. Granted, it is very unlikely you could be assured Word is not open given the scenario in this chapter.

The class now has the perfect constructor method : New (at least in its eyes). I bet you are wondering how this guy deconstructs aren't you? Me too. Let's fiddle with the Dispose method next.

Dispose

Because the form's New method contains customized logic creating the Word application, the Dispose method is the proper location for the custom code needed to release the class's reference to Word. The method's definition is as follows:

```
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
  If disposing Then
    If Not (components Is Nothing) Then
      components.Dispose()
    End If
  End If

  'Begin Custom Code
  If m_WordIsRunning Then
    m_Doc.Close()
  Else
    m_Doc.Close()
    m_appW.Quit()
  End If
  'End Custom Code

  MyBase.Dispose(disposing)
End Sub
```

Not complex at all. Before closing Word, the method uses the flag set in the New method to determine if the DocumentProcessor started Word. If so,only the newly created document is closed and Word is left running:

```
If m_WordIsRunning Then
  m_Doc.Close()
```

Closing the document and keeping Word open restores Word to the state it was in prior to the DocumentProcessor taking control. If Word was started by the DocumentProcessor, both the document and Word application are closed and removed from memory:

```
Else
  m_Doc.Close()
  m_appW.Quit()
End If
```

The class cleans up pretty well for itself, don't you think? The real heavy-lifting, however, is in the following six methods that work directly with a specific Word document.

OpenDoc

The OpenDoc method opens the Word document by combining the DocumenProcessor's OpenPath and FileName properties and calling Word's Open function:

```
Public Function OpenDoc() As Boolean
 Try
  With m_appW
   .DisplayAlerts = W.WdAlertLevel.wdAlertsNone
   .Documents.Open(m_OpenPath & "\" & m_FileName)
   m_Doc = .ActiveDocument
   .DisplayAlerts = W.WdAlertLevel.wdAlertsAll
  End With
  Return True
 Catch ex As Exception
  Return False

 End Try
End Function
```

Before actually attempting to open the file, the function turns off Word's DisplayAlerts property. This avoids the situation where Word pops-up a message to the user by displaying a dialog box. This would only confuse the user so it is turned off while an attempt is made to open the document:

```
   .DisplayAlerts = W.WdAlertLevel.wdAlertsNone
   .Documents.Open(m_OpenPath & "\" & m_FileName)
```

If the document opens successfully, it is stored for use within the class and the alerts are reactivated:

```
   m_Doc = .ActiveDocument
   .DisplayAlerts = W.WdAlertLevel.wdAlertsAll
```

PopulateForm

We have an opened Word document ready for action. The next thing we need to do is populate the DocumentProcessor form with a series of TextBox controls representing each Bookmark in the document.

The PopulateForm method does just that, itreads the Bookmarks collection contained in the referenced document and creates corresponding Label and TextBox controls on the DocumentProcessor form. As each TextBox is created, it is filled with corresponding data from m_Contact. Take a look and then head on down for a deeper discussion:

```
Public Sub PopulateForm()
```

```
    Dim bm As W.Bookmark
    Dim i As Integer
    Dim iTop As Integer
    Dim itms As OL.ItemProperties
    Dim itm As OL.ItemProperty
    Dim val As String

    Try
     iTop = pnlTitle.Top + 100
     itms = m_Contact.ItemProperties

     For Each bm In m_Doc.Bookmarks
      Dim lbl As New Windows.Forms.Label
      Dim txt As New Windows.Forms.TextBox

      With lbl
       .Text = bm.Name
       .Left = 10
       .Width = 150
       .Top = iTop
       .Visible = True
      End With

      If PropertyExists(m_Contact, bm.Name) Then
       itm = itms(bm.Name)
       val = itm.Value
      Else
       val = ""
      End If

      With txt
       .Name = bm.Name
       .Text = val
       .Width = 250
       .Left = lbl.Width + 10
       .Top = iTop
       .Visible = True
      End With

      Me.Controls.Add(txt)
      Me.Controls.Add(lbl)

      iTop = iTop + 22
     Next
```

```
    Catch ex As Exception
      Throw New System.Exception("An exception has occurred.")
    End Try
  End Sub
```

The trick to this method is synchronizing the creation of the TextBox controls (and their data population) with the reading of the Bookmark items within the document. The first step is to do a little prep work. The method uses a counter variable to position each newly created control. The first control will be positioned 100 pixels below pnlTitle:

```
iTop = pnlTitle.Top + 100
```

> Tip: If you want the controls closer together or farther apart, simply adjust the number to meet your specifications.

To simplify the code, the properties of m_Contact are stored in an Outlook ItemProperties typed variable:

```
itms = m_Contact.ItemProperties
```

Now the method is ready to work. We have a ContactItem and we have a mechanism for position the controls. We are ready to read the Bookmarks and build the form. For each Bookmark in the Word document, a new Label and TextBox are dimensioned as follows:

```
For Each bm In m_Doc.Bookmarks
  Dim lbl As New Windows.Forms.Label
  Dim txt As New Windows.Forms.TextBox
```

Using the new Label control, its properties are set as follows:

```
  With lbl
  .Text = bm.Name
  .Left = 10
  .Width = 150
  .Top = iTop
  .Visible = True
    End With
```

The Text property is set equal to the Bookmark name. This will help the user understand what information to input into the corresponding TexBox. Before setting the properties of the TextBox, a quick check is performed to determine if the ContactItem has a property with the same name as the current Bookmark:

```
  If PropertyExists(m_Contact, bm.Name) Then
    itm = itms(bm.Name)
    val = itm.Value
  Else
    val = ""
  End If
```

If the corresponding property exists, the value is read and stored for use in the setup of the TextBox control:

```
With txt
.Name = bm.Name
.Text = val
.Width = 250
.Left = lbl.Width + 10
.Top = iTop
.Visible = True
End With
```

Notice that the TextBox is named after the current Bookmark. This is important later when the form attempts to update the document's Bookmark collection with the values entered by the user. With all the properties set, the Label and TextBox are added to the form, the counter is updated to the next position, and the loop moves to the next Bookmark item:

```
 Me.Controls.Add(txt)
 Me.Controls.Add(lbl)

 iTop = iTop + 22
Next
```

Upon successful completion of PopulateForm, the DocumentProcessor is ready to show it's self off to the user. In this project, the Documents form displays DocumentProcessor in the following lines:

```
Dim dp As New DocumentProcessor
  …
dp.ShowDialog()
```

This call is part of the Documents GenerateDocs method discussed in the previous section.

PropertyExists

One thing to be careful of anytime you are referencing an Outlook item's properties is the fact that the property may not exist. A neat and clean strategy for handling this possibility is to create a helper function to check for the existence of a specified property name. If the property exists, the function returns True, if the property does not exist, False is returned. You're in luck because this is exactly why PropertyExists, um, exists.

```
Private Function PropertyExists(ByVal Contact As OL.ContactItem, _
ByVal PropertyName As String) As Boolean

Dim prop As OL.ItemProperty
prop = Contact.ItemProperties(PropertyName)

If Not prop Is Nothing Then
```

```
    Return True
  Else
    Return False
  End If

End Function
```

Making great use of the passed Outlook ContactItem, PropertyExists attempts to retrieve a property that matching the passed PropertyName argument:

```
Dim prop As OL.ItemProperty
prop = Contact.ItemProperties(PropertyName)
```

If the property is not found the prop variable will contain Nothing. We test for this and return either true or false to specify if the property was found or not:

```
If Not prop Is Nothing Then
  Return True
Else
  Return False
End If
```

With that complete, let's focus our attention on these Bookmark thingys.

UpdateBookMarks

After the user inserts values into the form' Textboxes and hits the OK button, we need a mechanism for updating the bookmarks in the document. That's right, I said mechanism. In addition, we need to be careful and only process Textboxes.

The UpdateBookMarks method loops through the Controls collection of the DocumentProcessor form. If the control is a TextBox, the SetBookmark method is called:

```
Private Sub UpdateBookMarks()
  Dim ctl As Frm.Control

  For Each ctl In Me.Controls
    If TypeOf (ctl) Is Frm.TextBox Then
      SetBookmark(ctl)
    End If
  Next

End Sub
```

With a simple check of each controls type, we avoid the nasty situation of accidentally passing a non-Textbox type control to SetBookMark.

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

SetBookMark

SetBookMark accepts a TextBox control as an argument and uses it to fill the BookMark of the same name of the TextBox within the Word document:

```
Private Sub SetBookMark(ByVal txt As Frm.TextBox)
  Dim rng As W.Range
  Dim str As String
  str = txt.Name.ToString

  If m_Doc.Bookmarks.Exists(str) Then
    rng = m_Doc.Bookmarks(str).Range
    rng.InsertBefore(txt.Text)
  End If

End Sub
```

Before inserting any text into the Bookmarks Range object, the method checks to make sure the Bookmark exists. If it does exits, the text within txt is inserted using the InsertBefore method of the BookmarkIf a particular string needs to be inserted throughout the document, the best strategy is to create one bookmark in the document and then insert REF fields anywhere the bookmark value should be repeated. This avoids a situation where a document contains several bookmarks containing the same information. To insert a REF field, select Insert➢Fields… from the main menu, REF from the Field Names listing, and then select the desired bookmark from the Field Properties listing (not applicable, though, if the referenced bookmark is actually destroyed)

> Caution: When updating the Text property of bookmark's Range object, understand that the bookmark will be destroyed. This is okay as long as they are not needed in the future. However, ifthey may be required, they can easily be created at the time the Text is updated by calling the Add method of the Bookmarks collection.

> Tip: To avoid any trouble, insert a space character (chr 32) into each Bookmark in the document templates. Then insert with REF fields for duplicating the Bookmark as needed. In code, call InsertBefore method of the Bookmark to insert text at the beginning of the Bookmark Range. This will leave the Bookmark intact for future use.

SaveDoc

Thanks to our two previous "Bookmark" methods, we now have an updated document filled with custom values. All that's left to make this a whiz bang class is to actually save the document somewhere. In this case, the SaveDoc method will save the updated document to the folder specified in the class's SavePath Property:

```
Private Function SaveDoc() As String
 Try
  m_appW.ActiveDocument.Fields.Update()
  m_appW.ActiveDocument.SaveAs(m_SavePath & "\" & _
   m_Contact.CompanyName & " - " & m_FileName)

  m_SavedPathAndFileName = m_Contact.CompanyName & " - " & m_FileName
  m_DocSaved = True

 Catch ex As Exception
  m_DocSaved = False
 End Try

End Function
```

In case the document contains any REF fields linked to bookmarks, the method calls the Field collection's Update method:

```
m_appW.ActiveDocument.Fields.Update()
```

Next, the document is saved to the specified save location, m_SavedPathAndFileName is updated with the full path and filename of the saved document and DocSaved flag is set to True:

```
m_appW.ActiveDocument.SaveAs(m_SavePath & "\" & _
 m_Contact.CompanyName & " - " & m_FileName)

m_SavedPathAndFileName = m_Contact.CompanyName & " - " & m_FileName
m_DocSaved = True
```

If any error occurs along the way, the DocSaved flag is set to False:

```
Catch ex As Exception
 m_DocSaved = False
```

Everything's all set, almost. All that's left to complete the class is to call the appropriate methods from the form's two Button's.

Form and Control Events

cbOK_Click

The OK Button is the spark that kicks everything off. To get things going, it calls the UpdateBookMarks method. It then saves the document and closes the form.

```
Private Sub cbOK_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cbOK.Click

 UpdateBookMarks()

 Me.SaveDoc()
```

```
    Me.Close()
End Sub
```

cbCancel_Click

The Cancel Button isn't exactly exciting but it would really annoy the user if we didn't have it. So here it is:

```
Private Sub cbCancel_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cbCancel.Click


    Me.Close()
End Sub
```

Okay, now we're done with the DocumentProcessor. We covered a lot with this class so here is a quick recap of is everything it can do:

* Open and Close Word
* Open a document
* Read the document's Bookmarks and dynamically fill the form with Labels and Textboxes
* Update the document Labels using the values in the Textboxes
* Save the document

Although this completes all the new material covered in this chapter, we have two more classes to cover – Settings and GetupDates. Let's take a quick run through them while highlighting any differences from Chapter 2.

The Settings Form

The Settings form is a copy of the version from Chapter 2 but has been slightly modified.. In addition to specify folder location and database settings, this version captures the location of an Outlook folder. This allows the user to specify the location of the folder containing Bravo Corp Client contact items.

Figure 3- illustrates the modified version of the Settings form. The modifications require a GroupBox control (gbOutlook), a Label control (lblFolderID), a TextBox control (txtClientsFolder), and a Button control (cbPickFolder).

Insert 01973f0309.tif

Figure 3-9. The Modified Settings Form

Variable Declarations

The form now has two class-level variables:

```
Private entryID As String
Private storeID As String
```

These variables are updated with upon selection to store the selected folder's MAPI data store EntryID and StoreID.

Methods

The SaveSettings and LoadSetting methods are slightly modified to include the new settings.

SaveSettings

```
Private Function SaveSettings()
  Try
    UserSettings.TemplatesFolder = txtTemplates.Text
    UserSettings.SaveFolder = txtSave.Text
    UserSettings.DatabaseName = txtDatabase.Text
    UserSettings.ServerName = txtServer.Text
    UserSettings.UserName = txtUserID.Text
    UserSettings.Password = txtPassword.Text
    UserSettings.ClientsFolderPath = txtClientsFolder.Text
    UserSettings.EntryID = EntryID
    UserSettings.StoreID = StoreID
    UserSettings.SaveSettings()
  Catch ex As Exception
    MsgBox(ex.Message)
  End Try

End Function
```

LoadSettings

```
Private Function LoadSettings()
  Try
    txtTemplates.Text = UserSettings.TemplatesFolder
    txtSave.Text = UserSettings.SaveFolder
    txtDatabase.Text = UserSettings.DatabaseName
    txtServer.Text = UserSettings.ServerName
    txtUserID.Text = UserSettings.UserName
    txtPassword.Text = UserSettings.Password
    txtClientsFolder.Text = UserSettings.ClientsFolderPath
    entryID = UserSettings.EntryID
    storeID = UserSettings.StoreID
  Catch ex As Exception
    MsgBox(ex.Message)
```

```
    End Try
End Function
```

Events

A single event is needed for the cbPickFolder event:

```
Private Sub cbPickFolder_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cbPickFolder.Click

 Dim str() As String

 str = appOutlook.PickContactsFolder()
 txtClientsFolder.Text = str(0)
 EntryID = str(1)
 StoreID = str(2)

End Sub
```

The appOutlook.PickContactsFolder method displays a dialog box (Figure 3-5) allowing the user to select an Outlook folder. The functions return a string array containing the Folder Path, MAPI EntryID, and MAPI StoreID. The method uses the values of the returned string to update txtClientsFolder with the FolderPath while EntryID and StoreID are stored within the class.

The UserSettings Class

The UserSettings class is,again, aslightly modified copy of the version included with the Presentation Generator of Chapter 1. This class handles the task of saving and loading the user settings specified in the Settings form. I will not give a full discussion of this class here except to explain the new code required to handle the saving of the Outlook folder information of that specified as: the Bravo Clients folder.

Three properties need to be saved from the Outlook folder selected in the Settings form: the FolderPath, EntryID, and StoreID. The FolderPath is used for the user's benefit and is cosmetic only. The latter two properties are what really matter to the add-in as together they identify the specified folder when calling appOutlook.GetFolder.

Variable Declarations

The following variables are added to the declarations section:

```
Private Shared m_strEntryID As String
Private Shared m_strStoreID As String
Private Shared m_strClientsFolderPath As String
```

In addition, the SETTINGS_XML_FILE_NAME constant needs a different value:

```
Private Const SETTINGS_XML_FILE_NAME As String = "\docgen.xml"
```

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

Property Procedures

The new properties for the UserSettings class are described in the following section.

ClientsFolderPath

The ClientsFolderPath property is a read/write property representing the name and path within the Outlook folder containing the Bravo Corp client records:

```
Public Shared Property ClientsFolderPath() As String
  Get
    Return m_strClientsFolderPath
  End Get
  Set(ByVal Value As String)
    m_strClientsFolderPath = Value
  End Set
End Property
```

EntryID

The EntryID property is a read/write property representing the unique EntryID of the folder specified by ClientsFolderPath:

```
Public Shared Property EntryID() As String
  Get
    Return m_strEntryID
  End Get
  Set(ByVal Value As String)
    m_strEntryID = Value
  End Set
End Property
```

StoreID

The StoreID property is a read/write property representing the unique EntryID of the folder specified by ClientsFolderPath:

```
Public Shared Property StoreID() As String
  Get
    Return m_strStoreID
  End Get
  Set(ByVal Value As String)
    m_strStoreID = Value
  End Set
End Property
```

Methods

The UserSettings class's two methods have each been updated to save and retrieve the three new properties.

SaveSettings

The SaveSettings method saves the specified add-in user settings to an XML file on the user's system. Three lines have been added to the method to save the new properties of the UserSettings class:

```
Public Shared Function SaveSettings() As Boolean
 Try
  Dim strPath As String
  strPath = strPath.Concat(UserSettings.SettingsPath, SETTINGS_XML_FILE_NAME)
  Dim xtwSettings As New XmlTextWriter(strPath.ToString, _
   System.Text.Encoding.UTF8)

  With xtwSettings
   .Formatting = Formatting.Indented
   .Indentation = 2
   .QuoteChar = """"c

   .WriteStartDocument(True)
   .WriteComment("User Settings from the Bravo Powerpoint Add-In")
   .WriteStartElement("BravoPowerPointUserSettings")
   .WriteAttributeString("UserName", UserSettings.UserName)
   .WriteAttributeString("Password", UserSettings.Password)
   .WriteAttributeString("SaveFolder", UserSettings.SaveFolder)
   .WriteAttributeString("TemplatesFolder", UserSettings.TemplatesFolder)
   .WriteAttributeString("ServerName", UserSettings.ServerName)
   .WriteAttributeString("DatabaseName", UserSettings.DatabaseName)
   .WriteAttributeString("ClientsFolderPath",UserSettings.ClientsFolderPath)
   .WriteAttributeString("EntryID", UserSettings.EntryID)
   .WriteAttributeString("StoreID", UserSettings.StoreID)
   .WriteEndElement()

   .WriteEndDocument()
   .Close()
  End With

  Return True
 Catch ex As Exception
  Return False
 End Try
End Function
```

LoadSettings

The LoadSettings method loads the specified user settings from an XML file on the user's system. Three lines have been added to the method to load the new properties values into the UserSettings class:

```
Public Shared Function LoadSettings(ByVal FilePath As String) As Boolean
  Try
    Dim strPath As String
    m_strSettingsPath = FilePath
    strPath = strPath.Concat(FilePath, SETTINGS_XML_FILE_NAME)
    Dim xtrSettings As New XmlTextReader(strPath.ToString)

    With xtrSettings
      .MoveToContent()
      m_strTemplatesFolder = .GetAttribute("TemplatesFolder")
      m_strSaveFolder = .GetAttribute("SaveFolder")
      m_strDatabaseName = .GetAttribute("DatabaseName")
      m_strServerName = .GetAttribute("ServerName")
      m_strUserName = .GetAttribute("UserName")
      m_strPassword = .GetAttribute("Password")
      m_strClientsFolderPath = .GetAttribute("ClientsFolderPath")
      m_strEntryID = .GetAttribute("EntryID")
      m_strStoreID = .GetAttribute("StoreID")
      .Close()

      Return True
    End With
  Catch ex As Exception
    Return False
  End Try
End Function
```

The GetUpdates Form Class

The GetUpdates form is exactly the same as the version in Chapter 1 with the exception that it downloads binaries from a different. For the sake of completeness the modified DownloadTemplates method of the GetUpdates form is as follows:

```
Private Function DownloadTemplates() As Boolean
  Dim cnn As New SqlConnection
  Dim da As New SqlDataAdapter("Select * From tblDocuments", cnn)
  Dim ds As New DataSet
  Dim dt As DataTable
  Dim drRecord As DataRow
  Dim btBinary() As Byte
```

This is a sample chapter of the forthcoming book: Office 2003 Programming: Real World Applications by Ty Anderson, ISBN: 1-59059-139-9

****

```
    Dim iSize As Long
    Dim strCnn As String

    Try
      strCnn = "Server=" & UserSettings.ServerName
      strCnn = strCnn.Concat(strCnn, ";uid=" & UserSettings.UserName)
      strCnn = strCnn.Concat(strCnn, ";pwd=" & UserSettings.Password)
      strCnn = strCnn.Concat(strCnn, ";database=" & UserSettings.DatabaseName)

      cnn.ConnectionString = strCnn.ToString
      cnn.Open()
      'Fill the Dataset using the SQLDataAdapter
      da.Fill(ds, "tblDocuments")


      'Loop through all records and save to the Default Save Location
      For Each drRecord In ds.Tables("tblDocuments").Rows
        btBinary = drRecord("DocumentBinary")

        Dim strPath As String = UserSettings.TemplatesFolder
        strPath = strPath.Concat(strPath, drRecord("DocumentName").ToString)

        Dim fsFile As New FileStream(strPath, FileMode.Create)
        iSize = UBound(btBinary)
        fsFile.Write(btBinary, 0, iSize)
        fsFile.Close()
        fsFile = Nothing
        strPath = Nothing
      Next drRecord

      Return True

    Catch ex As Exception
      Return False
    Finally

      cnn.Close()
      drRecord = Nothing
      ds = Nothing
      dt = Nothing
      da = Nothing
      cnn = Nothing
    End Try
  End Function
```

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

The Connect Class

The Connect class for the Document Generator is a skeleton of the version discussed in Chapter 1. This version of the Connect class really one serves to provide the IDTExtensibility2 interface and initialize any classes required for the add-in to function properly at startup. Of the five methods required to implement the interface, only two have any code within them.

OnConnection

The OnConnection event looks like this:

```
Public Sub OnConnection(ByVal application As Object, _
 ByVal connectMode As Extensibility.ext_ConnectMode, _
 ByVal addInInst As Object, ByRef custom As System.Array) _
 Implements Extensibility.IDTExtensibility2.OnConnection

 applicationObject = application
 addInInstance = addInInst
 UserSettings.LoadSettings(System.Windows.Forms.Application.StartupPath)
 appOutlook.Setup(application, UserSettings.EntryID)

End Sub
```

After references are set to the Outlook host application and the Document Generator add-in, the method loads the add-in's user settings and initializes the shared appOutlook class.

OnDisconnection

The OnDisconnection method shuts down the add-in as follows:

```
Public Sub OnDisconnection(ByVal RemoveMode As _
 Extensibility.ext_DisconnectMode, ByRef custom As System.Array) _
 Implements Extensibility.IDTExtensibility2.OnDisconnection

 appOutlook.ShutDown()
 UserSettings.SaveSettings()

End Sub
```

Summary

That's it for now. We have achieved quite a lot with this little add-in. In this chapter you learned how to automate the process of Word document creation through the utilization of Word Bookmarks. In addition, you learned how to combine this strategy with the functionality of Outlook to create a familiar user interface. Lastly, you learned a strategy

THIS IS A SAMPLE CHAPTER OF THE FORTHCOMING BOOK: OFFICE 2003 PROGRAMMING: REAL WORLD APPLICATIONS BY TY ANDERSON, ISBN: 1-59059-139-9

****

for saving the new generated documents to the user's system and then creating an Outlook MailItem containing the documents as attachments.