Click & Retrieve Source CODE!

Creating OLE Servers

Using VB4 to create an OLE server is easy, if you follow a few tips and tricks.

by Deborah Kurata

One of the exciting things you can do with classes is create OLE servers. OLE servers expose your application's objects and their properties and methods to other applications. You can develop a centralized inventory-control OLE server, for example, and any application that needs to adjust inventory can simply use your precompiled component. Or you can write a pricing-model server, or a to-do-list server, or an API wrapper, or a standard login screen, or … the possibilities are endless!

Using Visual Basic 4.0 to create an OLE server is easy, if you know a few tips and tricks. Once you've created the OLE server, any application that supports OLE Automation (called an OLE *client* application) can use it, including applications developed in Visual Basic 3.0 or 4.0, Visual C++, Excel, and Access.

TIP 1: DESIGN THE OLE SERVER

This tip seems obvious enough, but is frequently overlooked or abbreviated. The trick to a good design is first to clearly identify the objects your server will provide and the properties and methods of those objects. An easy way to do this is with CRC cards, which are simply index cards used to document your object design (see Programming with Class, "Designing Objects for VB4," by Kathleen Dollard-Joeris, in the December 1995 issue of *VBPJ*).

After you have identified the objects, you need to establish the relationships between the objects. This involves creating an object hierarchy for an inventory-control OLE server (see Figure 1). Be sure to have one object at the top of the hierarchy. If you have more than one at the top, you will need to define an additional object that will reside above the other objects at the top of the hierarchy. This top-level object will own the other objects in the application. If there is more than one top-level object, it is not clear which object owns the others.

To keep your code simple, keep the hierarchy relatively flat. Don't add too many levels to your design. Because Visual Basic 4.0 supports containment ("has a") relationships instead of inheritance ("is a") relationships, each higher-level object must expose the lower-level objects. If the hierarchy gets over three or four levels high, managing the composition relationships becomes much more difficult.

Another trick to OLE server design is good naming conventions. Because the OLE server can be used from any other application that supports OLE Automation (including the executives' Excel spreadsheets), you want to give the properties and methods of the OLE server good names. In an inventory-control OLE server, for example, instead of a property with a techie name such as gsInvProdName, the property could simply be ProductName. Instead of a method called iAddInv, the method could simply be AddInventory.

TIP 2: SEPARATE FORM AND FUNCTION

During the design stage, think carefully about the functionality of your OLE server separate from any user interface. In most cases, your OLE server does not need its own user interface. Rather, a client application can provide the interface and the OLE server can provide the functionality required for that interface.

For example, in developing an inventory-control OLE server, you'll provide methods for incrementing inventory when products are received, decrementing inventory when products are sold, and adjusting inventory when physical counts are taken. The OLE server could provide screens for each of these functions, but a better implementation allows the OLE client applications to control the user interface. You can then easily change, localize, port, or replace the user interface without affecting the OLE server. Likewise, you can modify the business rules in the OLE server without affecting the client application's user interface.

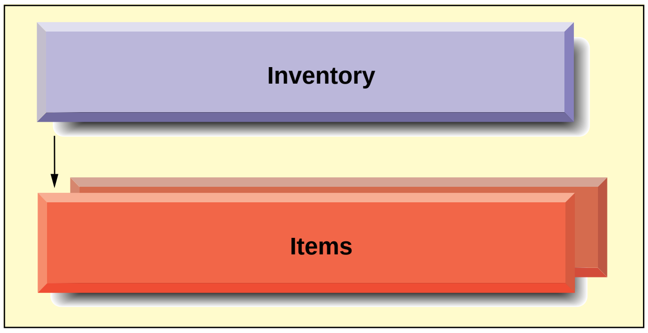
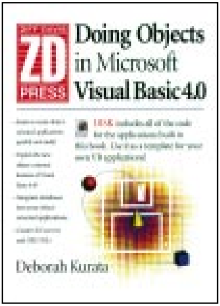


FIGURE 1 ***Drawing the Object Hierarchy.*** *The object hierarchy depicts the containment, or composition, relationships. In case, the inventory "has a" set of items.*

Deborah Kurata is principal consultant and cofounder of InStep Technologies, a consulting group that designs and develops object-oriented Microsoft Windows applications. She is the author of Doing Objects in Microsoft Visual Basic 4.0*, published by Ziff-Davis Press, which focuses on a pragmatic approach to object-oriented design and development of Visual Basic applications. Reach her at InStep Technologies, 5424 Sunol Blvd. #10-229, Pleasanton, CA 94566; or on CompuServe at 72157,475. You can also find her leading the Beginner's Corner section of the VBPJ Forum on CompuServe.*



This is even more important if you are planning to provide the OLE server as a remote automation server, a server installed on a system separate from the OLE client applications. If the OLE server does have a user interface, it will appear to the system it is installed on. So if your Receive Inventory form is part of the OLE server, the form will appear on the remote server computer instead of the local computer requesting the OLE server.

TIP 3: PROTOTYPE THE SERVER

Once the design of the server is complete, the next logical (and fun) step is to prototype the server. This step involves actually writing the outline of the code for the server. You can create the class modules, add the Property procedures, and define the subroutines and functions for the methods. Because this is a prototype, you don't need to develop any real code to access the data or process information. You can simply hard-code values in the Initialize events and add comments where the *real* code will need to go.

For the inventory-control OLE server, an Item class will track the information required for each item in inventory. As with every class, begin the code with a header, meeting the coding standards you follow:

```
' Class Name:     CItem
' Author:         Deborah Kurata, InStep
                  Technologies
' Date:           December 10, 1995
' Description:    Track inventory item
' Revisions:
' NOTE:
'  This is currently a prototype!
```

The Declarations section of the class contains the definition of the properties of this class. Define these properties to be private and local to this class. They will be exposed publicly through the Property procedures:

```
' Private data members
Private m_lProductID As Long
Private m_sProductName As String
Private m_sProductNumber As String
Private m_iQuantity As Integer
```

VB4

```
Private Sub Class_Initialize()
  ' Create the collection of inventory
  ' items
  CollectProducts
End Sub

' Fills the collection with the products
Private Sub CollectProducts()
  ' This is the prototype code
  ' to create the data in the collection
  ' Normally this information would
  ' exist in a database
  Dim Item As New CItem

  Item.ProductID = "1"
  Item.ProductNumber = "WD-123"
  Item.ProductName = "Widgets"
  Item.Quantity = 16

  ' Add this one to the collection
  ' Using the product number
  ' as the key
  m_colItems.Add Item, _
     Item.ProductNumber

  ' Clear the reference
  Set Item = Nothing

  ' Set another one
  Item.ProductID = "2"
  Item.ProductNumber = "GG-123"
  Item.ProductName = "Gagets"
  Item.Quantity = 120

  ' Add this one to the collection
  ' Using the product number
  ' as the key
  m_colItems.Add Item, _
     Item.ProductNumber

End Sub
```

LISTING 2 ***Initialize It.*** *Notice the trick with the Set Item = Nothing statement in the CollectProducts routine within the Initialize event. This clears the object reference in preparation for creating another object. Without this statement, the new assignments would simply be updating the prior object.*

VB4

```
' Removes inventory from the system
' This is used when inventory is sold,
' damaged, or adjusted
' Parameters:
'sInvItem    Product Number of the
'            item
'iNumRemoved    Number of the
'               product removed from
'               inventory
' Returns:
'iTotal      Total number in
'            inventory
Public Function _
  RemoveInventory(sInvItem As _
  String, iNumRemoved As Integer) As _
  Integer
Dim iCurrentTotal

  ' Retrieve the current inventory
  ' count from the collection
  iCurrentTotal = _
     Me.Item(sInvItem).Quantity

  ' Perform the subtraction
  iCurrentTotal = iCurrentTotal - _
     iNumRemoved

  ' Code here should ensure the total
  ' number in inventory is never < 0

  ' Code here could also check the
  ' reorder point and automatically
  ' reorder when an order point is
  ' reached

  ' Update the collection
  Me.Item(sInvItem).Quantity = _
     iCurrentTotal

  ' Return the total amount
  RemoveInventory = iCurrentTotal

End Function
```

LISTING 1 ***Take Them Away.*** *The RemoveInventory method of the inventory-control OLE server retrieves the current inventory count for the defined inventory item, decrements the total, updates the inventory count, and returns the current number of the item.*

The prototype of the Property procedures simply assigns and retrieves the value of the private data members. In the real application, you would add code to these to validate the data before assigning the value and to format the data before retrieving the value. A trick for making these easier to find later is to put them in alphabetical order:

```
Public Property Let ProductID(lID As Long)
   m_lProductID = lID
End Property
Public Property Get ProductID( ) As Long
   ProductID = m_lProductID
End Property
```

VB4

```
' Form Name:    frmInventory
' Author:       Deborah Kurata, InStep
'               Technologies
' Date:         December 10, 1995
' Description:  Inventory form.
' This is a little test form for the OLE
' server and NOT a good design for an
' inventory form!
'
' Revisions:
'
Option Explicit

' Private member variables
Private m_Inventory As CInventory

' Constants for the command butons
Const iCLOSE = 0
Const iRECEIVE = 1
Const iSELL = 2

Private Sub cboItems_Click()
  ' If the user selects an item
  ' clear the number received
  txtReceived.Text = ""

  ' Set the total in inventory as
  ' appropriate
  If cboItems.ListIndex <> -1 Then
     txtTotal.Text = m_Inventory.Item_
        (cboItems.ListIndex + 1).Quantity
  End If

  ' Ensure the button is disabled
  ' until a quantity is entered
  cmdInventory(iRECEIVE).Enabled = False
  cmdInventory(iSELL).Enabled = False
End Sub

Private Sub cmdInventory_Click(Index As Integer)
Dim iTotal As Integer
Dim sProductNumber As String

  Select Case Index

     Case iCLOSE
        Unload Me

     Case iRECEIVE
        ' Product number for the
        ' inventory item
        ' NOTE: combo's are 0 based,
        ' collections are 1 based
        sProductNumber = _
           m_Inventory.Item_
           (cboItems.ListIndex + 1).ProductNumber

        ' Add the defined amount to
        ' the inventory
        iTotal = m_Inventory.AddInventory_
           (sProductNumber, Val(txtReceived.Text))
        If Err.Number <> 0 Then
           ' A more friendly error
           ' message would be
           ' displayed to the user
           ' here
           ' With a more detailed
           ' message printed to a
           ' log file
           MsgBox Err.Description
        End If

        ' Display the revised
        ' inventory amount on the
        ' screen
        txtTotal.Text = iTotal

     Case iSELL
        ' Product number for the
        ' inventory item
        ' NOTE: combo's are 0 based,
        ' collections are 1 based
        sProductNumber = _
           m_Inventory.Item_
           (cboItems.ListIndex + 1).ProductNumber

        ' Remove the defined amount
        ' from the inventory
        iTotal = m_Inventory.RemoveInventory_
           (sProductNumber, Val(txtReceived.Text))
        If Err.Number <> 0 Then
           ' A more friendly error
           ' message would be
           ' displayed to the user
           ' here
           ' With a more detailed
           ' message printed to a
           ' log file
           MsgBox Err.Description
        End If

        ' Display the revised
        ' inventory amount on the
        ' screen
        txtTotal.Text = iTotal
  End Select
End Sub

Private Sub Form_Load()
Dim colItems As New Collection
Dim lCount As Long

  ' Create only one instance of the
  ' inventory class
  Set m_Inventory = New CInventory
  If Err.Number <> 0 Then
     ' A more friendly error message
     ' would be displayed to the user
     ' here
     ' With a more detailed message
     ' printed to a log file
     MsgBox Err.Description
     Unload Me
     GoTo EXIT_Form_Load
  End If

  ' Fill the combo box with values
  For lCount = 1 To m_Inventory.Count
```

CONTINUED ON PAGE 84.

LISTING 3 ***Tracking Your Inventory.*** *The code for this inventory-control test form would create the top-most level object in your object hierarchy, then exercise the properties and methods provided by the OLE server classes. The majority of this code responds to the Click event of the command buttons on the form.*

```
Public Property Let ProductName(sName As String)
   m_sProductName = sName
End Property
Public Property Get ProductName( ) As String
   ProductName = m_sProductName
End Property

Public Property Let ProductNumber(sNumber As String)
   m_sProductNumber = sNumber
End Property
Public Property Get ProductNumber( ) As String
   ProductNumber = m_sProductNumber
End Property

Public Property Let Quantity(iQuantity As Integer)
   m_iQuantity = iQuantity
End Property
Public Property Get Quantity( ) As Integer
   Quantity = m_iQuantity
End Property
```

CONTINUED FROM PAGE 82.

```
      With m_Inventory.Item(lCount)
         cboItems.AddItem .ProductNumber & _
            Space(5) & .ProductName
      End With
   Next lCount

 EXIT_Form_Load:
 End Sub

 Private Sub Form_Unload(Cancel As Integer)
   ' Clear the instance
   Set m_Inventory = Nothing
 End Sub

 Private Sub txtReceived_Change()
   ' If there is an amount entered and
   ' the command button is disabled,
   ' enable it
   If txtReceived.Text <> "" And _
      cmdInventory(iRECEIVE).Enabled _
      = False Then
      cmdInventory(iRECEIVE).Enabled _
         = True
      cmdInventory(iSELL).Enabled _
         = True
   End If
 End Sub

 Private Sub _
   txtReceived_KeyPress(KeyAscii _
   As Integer)
   If (Chr$(KeyAscii) <> vbBack) And _
      (Not IsNumeric(Chr$(KeyAscii))) _
      Then
      Beep
      KeyAscii = 0
   End If
 End Sub
```

The Inventory class owns the collection of inventory items. Define this containment relationship in the code by declaring the collection within the Inventory class:

```
' Class Name:  CInventory
' Author:      Deborah Kurata, InStep
               Technologies
' Date:        December 10, 1995
' Description: Track inventory
' Revisions:
' NOTE:
' This is currently a prototype!

Option Explicit

' Private data members
Private m_colItems As New Collection
```

The AddInventory method retrieves the current inventory count for the defined inventory item, increments the total, updates the inventory count, and returns the current number of the item:

```
' Add inventory when product is
' received
' Parameters:
' sInvItem          Product Number
' of the item
' iNumReceived    Number of the
' product received
' Returns:
' iTotal          Total number in
' inventory
Public Function AddInventory(sInvItem _
   As String, iNumReceived As _
   Integer) As Integer
Dim iCurrentTotal

   ' Retrieve the current inventory
   ' count from the collection
   iCurrentTotal = _
      Me.Item(sInvItem).Quantity

   ' Perform the addition
   iCurrentTotal = iCurrentTotal + _
      iNumReceived

   ' Update the collection
   Me.Item(sInvItem).Quantity = _
      iCurrentTotal

   ' Return the total amount
   AddInventory = iCurrentTotal
End Function
```

You could enhance the AddInventory code to ensure the amount in inventory does not exceed the storage capacity for that item. It could also generate receipt information, perform accounting transactions, and so on.

Because the collection of inventory items is private to this class, other applications cannot access the methods or properties of the collection. To provide access to the method or property, you can develop wrappers. The Count Property procedure in this class is simply a wrapper for the Count property of the collection and provides the current number of inventory items in the collection:

```
' Wrapper for collection count
Public Property Get Count() As Integer
   Count = m_colItems.Count
End Property
```

The Item method of this class is a wrapper for the Item method of the collection. It provides access to the lower-level object. This provides the mechanism for other applications to reference a property of the lower-level object, in this case the inventory item:

```
' Wrapper for the collection
' Parameters:
'  vIndex      string to find by key
'              numeric to find by
'              position
Public Function Item(ByVal vIndex As Variant) As CItem
   Set Item = m_colItems.Item(vIndex)
End Function
```

The RemoveInventory method retrieves the current inventory count for the defined inventory item, decrements the total, updates the inventory count, and returns the current

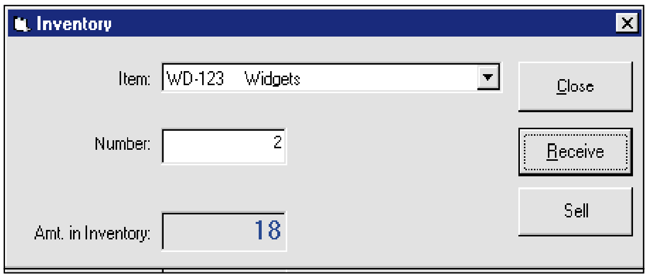


FIGURE 2 ***Inventory Test Form.*** *Test the operation of your OLE server with a simple, multipurpose form. This test form doesn't need to be part of the actual user interface of your application.*

number of the item (see Listing 1). You could add code to this method to ensure the inventory does not drop below zero for an item or to automatically define a back order. In addition, this could check a predefined reorder point for an item and automatically generate an order if the inventory count drops below the defined amount.

The Initialize event for the class would normally fill the collection of items from a database. For the prototype, this information is hard-coded into the CollectProducts routine (see Listing 2). Notice the trick with the Set Item = Nothing statement shown between the two assignments. This clears the object reference in preparation for creating another object. Without this statement, the new assignments would simply be updating the prior object. The collection would then contain multiple references to the same object. (Try this and then display the ProductID of each item in the collection. They will all be the same!)

Finally, don't forget to add that standard module with a Sub Main routine. This routine does not have to contain any code (the prototype didn't need anything here), but you need to define the Sub Main so the OLE server will start up.

OLE SERVERS EXPOSE YOUR APPLICATION'S OBJECTS AND THEIR PROPERTIES AND METHODS TO OTHER APPLICATIONS.

TIP 4: DEVELOP A STANDALONE APPLICATION

Once you have the prototype code in place, you will want to run your OLE server. But an OLE server serves up objects, so unless something is requesting objects, the server doesn't appear to do anything. You need to write an OLE client application that will create objects from your OLE server.

However, testing and debugging an OLE server is a lot harder than testing and debugging a normal standalone application. The trick here is to perform the first set of tests on the OLE server as if it were a normal standalone application.

To do this, simply add a form to the application that will act as the client for the server. Make this form the startup form for this application. For the inventory-control OLE server, I added a simple data-entry form (see Figure 2). It is often easier to test the server by providing access to all server functionality from one form. This test form need not be part of the actual user interface of your application.

The code for this inventory-control test form would create the top-most level object in your object hierarchy, then exercise the properties and methods provided by the OLE server classes (see Listing 3).

One of the benefits of developing a prototype and testing it in this manner is that you will find out, after only a few hours, if your design will provide the results you need. You will also learn about some of the rules of creating classes and accessing OLE servers. For example, you can't pass user-defined types (UDTs) as parameters to a method in a class module and you can't return a UDT from a method of a class module.

TIP 5: MAKE THE APPLICATION AN OLE SERVER

When you are confident of the basic operation of your classes, you'll want to make the application into a real OLE server. To do this, first remove the test form from the project and then set the Public property on each class that will expose objects to True. The tricky part here is to be sure to set all exposed contained classes to Public as well. So if you expose a lower-level object, the class for that lower-level object needs to have the Public property set to True as well. For example, in the inventory-control OLE server, the Item object is exposed through the Item method of the Inventory class, so the Item class must have the Public property set to True.

If you don't want any application creating objects from the lower-level contained classes, simply set the Instancing property of the lower-level classes to None. This will prevent any other application from creating objects from these lower-level classes. However, be sure to set the Instancing property of the top-most class to Creatable SingleUse or Creatable MultiUse. Otherwise the form will not be able to create any objects from your server.

After you've set the correct Public and Instancing properties for your classes, you can use the Make EXE File command to make an executable. This executable is your OLE server. Note that because this executable is a Visual Basic application, it still needs all of the associated DLLs such as VB40016.DLL or VB40032.DLL. Keep this in mind when you attempt to install this OLE server on another computer. The trick to installing the components is to use the SetupWizard to install your OLE server and in Step 6, to set the "Install as OLE Automation shared component" deployment model. This will ensure all correct files are located, compressed, and copied to the diskette, then installed correctly on another system.

Once you've created the EXE, you can use the Compatible OLE Server text box in the Project tab of the Options dialog box to set this EXE as a Compatible OLE Server. This will allow you to make changes to the server later without affecting the reference to the server. I'll discuss this in more detail in Tip 6.

If your application is ultimately going to be an OLE DLL instead of an EXE, you may want to follow this process to make it an EXE first. After testing the EXE, you can remake the application into an OLE DLL.

TIP 6: TEST, TEST, TEST

Now that you have a prototype of your OLE server, you can test it in all environments where it will ultimately be used. Will it be used from VB4, VB3, or Excel? If so, try all three to ensure the server works appropriately. Will it be used under Windows 3.1, Windows 95, or Windows NT? Again, try the server in each anticipated environment.

So how do you test the server? The easiest approach is to create a new project and add the form described in Tip 4 (see Figure 2). Before this test OLE client application will run successfully, you need to define where it will find the OLE server. Do this using the References option in the Tools menu. You can select the OLE server from the list of available references.

You can then run this OLE client application. The OLE server will be accessed appropriately. Notice that when you run the OLE server, it will appear in your system's task list. This is because the OLE server runs in its own separate task. To have the server run in the same process as the OLE client application, remake the OLE server using the Make OLE DLL command.

Now you are ready to add the *real* code to the OLE server and repeat these last three tips for running and debugging the server. Whether you're building an OLE EXE or OLE DLL, these tips and tricks should serve you well.