Chapter 5 – Alternate Content

The first edition of this book included a sample with linked lists to teach inheritance. For the second edition, I decided a more simple example that would be easier to follow would be better. However, the linked list example is included here for those who are interested.

The example we will use is that of a simple linked list. A linked list is an extremely useful structure for maintaining lists of objects. It can be more efficient than an array because you can insert and remove entries without shifting data in the array. It can be more useful than a collection because it is easier to control the order of objects in a linked list and rearrange them as needed.

The principle of a linked list is simple: each object contains a pointer (or reference) to the next object in the list. A root variable points to the first object in the list.

A VB6 Linked List

Let's begin by considering a VB6 class that implements a linked list. I know this is a book about VB.NET but trust me, the best way for you to really learn and understand inheritance is through comparison to containment and interface inheritance as they are implemented in VB6. The LinkListVB6 project demonstrates a class designed to make it easy to add linked list functionality to another class. In other words, the goal is to be able to reuse this code as easily as possible.

The LinkList object has the following properties and methods:

- Property NextItem—used to obtain a reference to the next object in the list.
- Property PreviousItem—used to obtain a reference to the previous object in the list.
- Method Remove—Removes the current object from a list.
- Method Append—Appends the current object to a list.

Under Visual Basic 6, you can achieve code reuse through containment. The class that you want to link can contain an instance of the LinkList class and can call its methods to perform the linking.

What type of object does the NextItem and PreviousItem properties refer to? Do they refer to a LinkList object or do they refer to the object that contains the LinkList object?

Figure 5-1 illustrates the first approach where each LinkList object can only point to another LinkList object. In this case, you need a way to obtain a reference to the container object, a hypothetical Customer object.

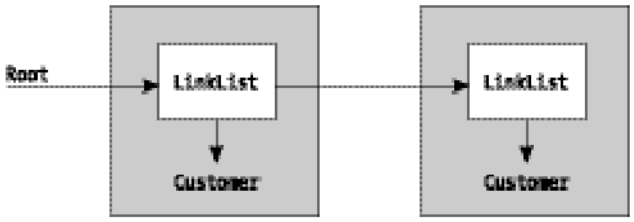

Figure 5-1. LinkList objects reference other LinkList objects. A Container property is used to obtain a reference to the container

Figure 5-2 illustrates the latter approach where each LinkList object points to the actual Container object that holds the internal LinkList object.

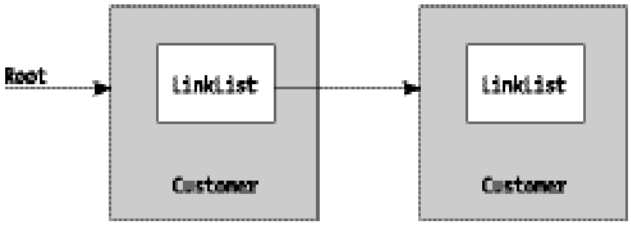

Figure 5-2. LinkList objects reference the Container objects.

Let's start with the latter approach, which is implemented by the LinkListVB6 project in the sample code for this book. The LinkList class is designed not only to be held by a container but also to provide the interface that the container can use to implement the linked list functionality. The Container object will use the Implements statement, which provides interface inheritance to quickly add linked list functionality to the object.

The LinkList object contains two Private variables. The m_Next variable is the reference to the next object in the list. The LinkList object will be pointing to the container but since the container implements the LinkList interface, the m_Next variable can be of type LinkList. The m_Container object holds a reference to the Container object. You'll soon see why it's needed. The LinkList object is defined
as follows:

```
' LinkList VB6 example #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit

' This version is designed to be embedded, so it links to the container
' object
Private m_Next As LinkList
' When adding to the list, it needs to know its container
Private m_Container As Object

' Container is set during initialization
' Would need to be public if you decided to componentize the class
Friend Property Set Container(ByVal ContainerObject As Object)
  Set m_Container = ContainerObject
End Property
```

The NextItem property simply returns a reference to the m_Next variable, which contains a reference to the next object in the list:

```
' Next is easy - just the next link.
Public Property Get NextItem() As LinkList
  Set NextItem = m_Next
End Property

Public Property Set NextItem(ByVal nextptr As LinkList)
  Set m_Next = nextptr
End Property
```

The PreviousItem property is a bit tricky. Since this is a singly linked list, the property needs a reference to the first object in the list so it can search forward for the current object or, rather, the object whose NextItem property is the current object. The tricky factor here is determining which object you are comparing to. You can't compare objects to "Me" because "Me" refers to the internal LinkList object, not the one referenced by the m_Next variable. No, the comparison must be to the Container objects; thus each LinkList object must hold a reference to its container as shown here:

```
' In a single linked list, Previous has to search forward from the root
Public Property Get PreviousItem(Root As LinkList) As LinkList
  Dim currentitem As LinkList
  ' Remember, all references are to the container object
  If (Root Is m_Container) Or (Root Is Nothing) Then
```

```
    Exit Property
  End If
  Set currentitem = Root
  Do
    If currentitem.NextItem Is m_Container Then
      Set PreviousItem = currentitem
      Exit Property
    Else
      Set currentitem = currentitem.NextItem
    End If
  Loop While Not currentitem Is Nothing
End Property
```

The Remove method uses the PreviousItem property to find the object that precedes the one being removed. If one exists, its m_Next variable is set to the current object's m_Next variable, effectively removing this entry from the linked list. If the object being removed is the first object in the list, the root variable must be set to reference the next object in the list. This is done by setting the root variable (which is called by reference) to the next object in the list. Don't be confused by the fact that it is set to a variable of type LinkList. Root references the container (that implements the LinkList interface) and not the internal object:

```
' Remove has to search from root to find the previous node.
' Root must be by reference so it can be cleared if this
' is the last object, or reset if the first object
Public Sub Remove(Root As LinkList)
  Dim previtem As LinkList

  Set previtem = PreviousItem(Root)

  If previtem Is Nothing Then
    Set Root = m_Next
  Else
    Set previtem.NextItem = m_Next
  End If
End Sub
```

The Append subroutine also uses the m_Container property in order to assign an object correctly into the list. The routine finds the last object in the list, then sets its m_Next variable to the container of the current object:

```
' Append searches from Root to the end of the list.
Public Sub Append(Root As LinkList)
  Dim currentitem As LinkList
  Set currentitem = Root
  If Root Is Nothing Then
    Set Root = m_Container
  Else
    While Not currentitem.NextItem Is Nothing
      Set currentitem = currentitem.NextItem
    Wend
    Set currentitem.NextItem = m_Container
  End If
End Sub
```

The Container class is defined in the Customer.cls file. It has a public CustomerName property that allows you to access the name of a customer. This class implements the LinkList object, thus inheriting the LinkList interface. It contains a private LinkList object called m_MyLinkList that is initialized during the Class_Initialize event. As you can see in Listing 5-1, each of the implemented functions simply calls into the contained LinkList object.

Listing 5-1. Customer.cls class for the LinkListVB6 project.

```
' LinkList VB6 example #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit

' This version Implements the LinkList - making
' its methods accessible via LinkList objects
Implements LinkList

Public CustomerName As String

' The internal LinkList object provides the functionality
Private m_MyLinkList As LinkList

Private Sub Class_Initialize()
  Set m_MyLinkList = New LinkList
  Set m_MyLinkList.Container = Me
End Sub
```

```
' This will never be called. Read text for how
' to get around this problem
Private Sub Class_Terminate()
  Debug.Print "Terminating customer " & CustomerName
End Sub

' Methods & properties just map to the LinkList members & properties
Private Sub LinkList_Append(Root As LinkList)
  m_MyLinkList.Append Root
End Sub


Private Property Set LinkList_NextItem(ByVal nextobject As LinkList)
  Set m_MyLinkList.NextItem = nextobject
End Property

Private Property Get LinkList_NextItem() As LinkList
  Set LinkList_NextItem = m_MyLinkList.NextItem
End Property

Private Property Get LinkList_PreviousItem(Root As LinkList) As LinkList
  Set LinkList_PreviousItem = m_MyLinkList.PreviousItem(Root)
End Property

Private Sub LinkList_Remove(Root As LinkList)
  m_MyLinkList.Remove Root
End Sub
```

The TestForm.frm module contains a text box for new customer names, a Command button to add a new customer, a listbox to display the list of customers, and another Command button to remove the customer selected in the listbox. The module contains a variable m_List, which contains a reference to the first Customer object in the list. Again, even though the m_List variable is of the type LinkList, it references the Customer object itself (through its LinkList interface) and not the internal LinkList object:

```
' LinkList VB6 example #1
' Copyright ©2000 by Desaware Inc.  All Rights Reserved

Option Explicit Dim m_List As LinkList
```

Under VB6, you can access an interface's methods or properties using a variable that represents the interface you wish to use. To obtain a list of objects, you need two variables:

a LinkList object that allows you to scan through the list using its NextItem property and a Customer object that allows you to read the CustomerName property. This idea—that each interface provides its own set of properties and that each object can expose multiple interfaces—is fundamental to COM. When you assign a variable of one type to another (as seen in the following Set currentcustomer = currentlist line), you are performing a QueryInterface operation that COM uses to navigate from one interface to another on an object. You'll soon understand why I stress this point. Here is the code for the Testfrm.frm form:

```
' Updates the list box
' Note the use of separate variables to access each interface
Private Sub UpdateList()
  lstCustomers.Clear
  Dim currentlist As LinkList
  Dim currentcustomer As Customer
  Set currentlist = m_List
  Do While Not currentlist Is Nothing
    Set currentcustomer = currentlist
    lstCustomers.AddItem currentcustomer.CustomerName
    Set currentlist = currentlist.NextItem
  Loop
End Sub
```

The cmdAdd_Click and cmdRemove_Click functions both show the same use of two types of variables to work with the object as you can see in Listing 5-2.

Listing 5-2. Code to add and remove link list entries.

```
Private Sub cmdAdd_Click()
  Dim newEntry As New Customer
  Dim newEntryll As LinkList
  newEntry.CustomerName = txtCustomerName.Text
  Set newEntryll = newEntry
  newEntryll.Append m_List
  UpdateList
End Sub

Private Sub cmdRemove_Click()
  Dim currentlist As LinkList
  Dim currentcustomer As Customer

  Set currentlist = m_List
  Do While Not currentlist Is Nothing
    Set currentcustomer = currentlist
    If currentcustomer.CustomerName = lstCustomers.Text Then
      currentlist.Remove m_List
      UpdateList
```

```
      Exit Sub
    End If
    Set currentlist = currentlist.NextItem
  Loop
  UpdateList
End Sub
```

What can we conclude from this example?

- It is possible using containment and interface inheritance to reuse functionality with a minimal amount of coding. The amount of code needed to implement this type of containment in the container (Customer) object is minimal.

- Part of the awkwardness of interface inheritance comes from the fact that you must explicitly switch between interfaces to access the methods of each interface. This is reflected by the extra variables and assignments in the test form.

By the way, have you noticed the one truly fatal flaw in this example? If not, try running it and watch the Immediate window to see what happens when you remove objects from the list. Got it?

- The need for the Contained object to reference the Container object and for the Container object to reference the Contained object means that you automatically have a circular reference!

If you were to actually use this approach, you would need to find a way to eliminate this circular reference. The most common way to do this is to have the Contained object raise an event whose parameter is an Object variable passed by reference. When the container receives the event, it must set the object parameter to Me. This provides the LinkList object with a reference to its container. The LinkList object can then use the container reference directly for comparisons or return it as a value. The container reference should be released immediately to eliminate the circular reference. The flaw with this approach is twofold: it's awkward and it's slow. Events are not early bound, thus performance with this approach can be seriously impaired.

A Containment-Based Linked List Using VB.NET

Let's begin by looking at a direct port of the previous example from VB6 to VB.NET in solution LinkListNet2. Along the way, you will see how the syntax of VB.NET has changed. I think you'll see that while different, the code is not difficult to understand. But I also think you'll see additional confirmation that a major port from VB6 to VB.NET is not something to undertake lightly.

The first change you'll see with VB.NET is that the concept of interface has been separated from that of a class (implementation of an interface). With VB6, adding methods to an object automatically defines the interface of the object. With VB.NET, if you want to define an interface that can be implemented or shared, you must explicitly define it as an interface. The interface, called ILinkList,[1] is defined in the file LinkList.vb shown here:

```
' LinkList .Net example using aggregation
' Copyright © 2001 by Desaware Inc. All Rights Reserved

' We can't implement a class - only an interface.
' so here's the interface
Public Interface ILinkList
   WriteOnly Property Container() As Object

   Property NextItem() As ILinkList

   ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList

   Sub Remove(ByRef Root As ILinkList)

   Sub Append(ByRef Root As ILinkList)
End Interface
```

An interface is nice but we need an implementation for it (which can be used by the Customer object) as well. Thus, the LinkList object implements the ILinkList interface. The m_Next variable points to the ILinkList interface—not to the LinkList object. Why? Because we're going to want this variable to reference the Container object just as it did in the VB6 example. The Container object will implement the ILinkList interface—*not* the LinkList object:

```
   Public Class LinkList
      Implements ILinkList
      ' This version is designed to be embedded, so it links to the _
        container object
```

1 Interfaces, by convention, always start with the letter I.

```
    Private m_Next As ILinkList
    ' When adding to the list, it needs to know its container
    Private m_Container As Object
```

The syntax for implementing functions of an interface is different in VB.NET. Instead of a method name like ILinkList_Container, the method declaration explicitly says which method is being implemented. The method name used in the class can be completely different from the interface method name. It is even possible for one method to implement multiple interface methods. Instead of separate property Get and property Set/Let methods, the Get and Set blocks are part of the Property definition itself.

A property Get block returns the value of the property by assigning a value to the property name (as with VB6 or by using the Return statement.). The property Set block can access the value being set using the built-in Value variable. Since there is no Set statement for object assignment in VB.NET, there is only one way to set a property value. Don't let the fact that it's called a Set block confuse you—they basically took the functionality of both of the VB6 Set and the Let property methods, combined them, and put them into the Set block.

The property syntax has changed as well:

```
' Next is easy - just the next link.
' Note the syntax change. See text for discussion of the Implements
' statement as used here
Public Property NextItem() As ILinkList Implements ILinkList.NextItem
    Get
        NextItem = m_Next
    End Get
    Set(ByVal Value As ILinkList)
        m_Next = Value
    End Set
End Property
```

In VB6, a property is specified as read only or write only by the presence or absence of the corresponding Get or Set property methods. In VB.NET, the attribute WriteOnly or ReadOnly is used to control the property access. Only the necessary block is included in the property implementation:2

---

2 Why do you need to specify ReadOnly or WriteOnly when the language should be able to figure this out based on whether you included Set or Get code? That's a good question. My best guess is that they could have figured it out from the code but the use of ReadOnly and WriteOnly fits better with the .NET architecture. The ReadOnly and WriteOnly keywords are actually attributes, which you'll learn about in great detail in Chapter 11.

```
' Container is set during initialization
' Would need to be public if you decided to componentize the class
Friend WriteOnly Property Container() As Object Implements ILinkList.Container
   Set(ByVal Value As Object)
      m_Container = Value
   End Set
End Property
```

Aside from the syntax changes already mentioned, the code for the PreviousItem property and Remove methods are very similar to that in VB6:

```
' In a single linked list, Previous has to search forward from the root
Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) As _
ILinkList Implements ILinkList.PreviousItem
   Get
      Dim currentitem As ILinkList
      ' Remember, all references are to the container object
      If (Root Is m_Container) Or (Root Is Nothing) Then
         Exit Property
      End If
      currentitem = Root
      Do
         If currentitem.NextItem Is m_Container Then
            PreviousItem = currentitem
            Exit Property
         Else
            currentitem = currentitem.NextItem
         End If
      Loop While Not currentitem Is Nothing
   End Get
End Property

' Remove has to search from root to find the previous node.
' Root must be by reference so it can be cleared if this
' is the last object, or reset if the first object
Public Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
   Dim previtem As ILinkList

   previtem = PreviousItem(Root)
   If previtem Is Nothing Then
      Root = m_Next
   Else
```

```
        previtem.NextItem = m_Next
      End If
   End Sub
```

The Append method is very similar to the VB6 edition except for two things: assigning the Root parameter from the m_Container variable and assigning the NextItem property from the m_Container variable. In VB6, this is a simple assignment. Here, it is necessary to explicitly convert the m_Container object to an ILinkList type. Why is this?

At first glance, you might think that this is an example of the improved Strict Type Checking provided by VB.NET. And to some degree this is true. But in fact, there is more going on here than meets the eye. Hold this question for just a few more paragraphs and I promise you an answer that will hopefully lead to some fundamental insights as to the difference between VB6 and VB.NET:

```
   ' Append searches from Root to the end of the list.
   Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
      Dim currentitem As ILinkList
      currentitem = Root
      If Root Is Nothing Then
         Root = CType(m_Container, ILinkList)
      Else
         While Not currentitem.NextItem Is Nothing
            currentitem = currentitem.NextItem
         End While
         currentitem.NextItem = CType(m_Container, ILinkList)
      End If
   End Sub

End Class
```

The Customer object is in file Customer.vb and is shown in Listing 5-3. The Customer object implements the previously defined ILinkList interface and has a public CustomerName property as in the VB6 example. Other than the VB.NET syntax changes, this class is identical to the VB6 Customer object.

Listing 5-3. The Customer class is implemented in file Customer.vb.

```
' LinkList .Net example using aggregation
' Copyright © 2001 by Desaware Inc. All Rights Reserved
Public Class Customer
```

```
' This version Implements the LinkList - making
' its methods accessible via LinkList objects
Implements ILinkList

Public CustomerName As String

' The internal LinkList object provides the functionality
Private m_MyLinkList As LinkList
Public Sub New()
   MyBase.New()
   m_MyLinkList = New LinkList()
   m_MyLinkList.Container = Me
End Sub

' Objects will terminate in VB.Net. See text for details.
Protected Overrides Sub Finalize()
   system.diagnostics.Debug.WriteLine("Terminating customer " + CustomerName)
End Sub

' Methods & properties just map to the LinkList members & properties
Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
   m_MyLinkList.Append(Root)
End Sub

Public Property NextItem() As ILinkList Implements ILinkList.NextItem
   Set(ByVal Value As ILinkList)
      m_MyLinkList.NextItem = Value
   End Set
   Get
      NextItem = m_MyLinkList.NextItem
   End Get
End Property

Friend WriteOnly Property Container() As Object Implements ILinkList.Container
   Set(ByVal Value As Object)
      m_MyLinkList.Container = value
   End Set
End Property

Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList __
Implements ILinkList.previousitem
   Get
      PreviousItem = m_MyLinkList.PreviousItem(Root)
```

```
    End Get
  End Property

  Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
    m_MyLinkList.Remove(Root)
  End Sub

End Class
```

Now let's look at the form code found in file TestForm.vb. I've excluded the code generated by the framework as it really isn't relevant to understanding the issue at hand. As with the VB6 example, a variable is defined to point to the first object in the list:

```
Public Class Form1

    Private m_List As ILinkList
```

The UpdateList subroutine loads the listbox with the list of current customers. There are a number of very important changes between the VB6 version and this one.

One change is small—listboxes don't work the way they used to. Instead of using an AddItem method, you must add the string to the Items collection of the listbox. Indeed, all of the Windows Forms controls[3] have syntactical and functional differences from their VB6 equivalents.

You can also see that this function has an explicit conversion of the m_List variable (which is of type ILinkList) to the currentcustomer object (which is of type Customer).

But the big change is that this function no longer has two different variables to access the object—one for the Customer interface and the other for the LinkList interface. The currentcustomer object can directly access the CustomerName property and the NextItem property as shown in Listing 5-4.4

Listing 5-4. Update function in TestForm.vb.

```
  ' Updates the list box
  ' Note that there is no need for separate variables - the
  ' ILinkList interface is seamlessly added to the object
  ' Note also the change to the ListBox syntax
  Private Sub UpdateList()
```

---

3 Windows Forms controls are the .NET term for all controls in the .NET Framework. Think of them as the .NET equivalent of VB intrinsic controls or ActiveX controls.

4 You can, if you wish, use the Private attribute to hide the methods that implement the interface and allow access to the interface methods only through interface variables.

```
    lstCustomers.Items.Clear()
    Dim currentcustomer As Customer
    ' However, explicit type conversions are needed when promoting
    ' a reference to an interface to the object. Runtime errors will occur
    ' if the object type is not correct.
    currentcustomer = CType(m_List, Customer)
    Do While Not currentcustomer Is Nothing
      lstCustomers.Items.Add(currentcustomer.CustomerName)
      currentcustomer = CType(currentcustomer.NextItem, Customer)
    Loop
  End Sub
```

So, let's consider both of these facts again:

- In VB6, a variable must match the interface being used to call methods on that interface. In VB.NET, an Object variable can access all methods of all of its interfaces directly.

- In VB6, when you assign a variable that references an interface on an object to that of another interface on an object, the assignment works directly. In VB.NET, you must explicitly convert the type from the implemented interface to that of the Container object or other implemented interface.

These are not minor language changes. They reflect a fundamental change in the underlying architecture that is essential to understand.

COM dictates the behavior of Visual Basic 6. Under COM, an interface pointer references an object. Each interface pointer exposes a set of methods. If you want to call a method on a different interface for an object, you must navigate to the other interface and call the method on that interface. Each interface corresponds to a Visual Basic type, thus you must assign an object to a variable of the correct type before you can call the methods for that type.

VB.NET is not based on COM so the old rules simply don't apply.

When you implement an interface in VB.NET, you are in effect inheriting an interface. That interface becomes part of the object—a subset of the object. If you assign an object reference to a variable with the type of the inherited interface, the assignment can be made directly. This is because VB.NET knows at compile time that the object implements the interface. For example, the following code is correct:

```
Dim il as ILinkList
Dim co as Customer
co = New Customer
il = co
```

The assignment works because VB.NET knows that the Customer object always implements the ILinkList interface.

But the reverse is not true. If you try to assign a Customer object from an ILinkList object reference, there is no way for VB.NET to know at compile time if the ILinkList object you assign is a Customer object. The ILinkList variable might point to a LinkList object or to some other arbitrary object that implements the ILinkList interface. The CLR will know at runtime what type of object is referenced by that variable so the assignment can be done at runtime—but since it is not guaranteed to work, the compiler will raise an error if you simply try an assignment such as this one directly:

```
Dim il as ILinkList
Dim co as Customer
il = New Customer
co = il
```

Instead, you must perform an explicit conversion using the generic CType conversion function as follows:

```
co = CType(il, Customer)
```

This tells the compiler that you think you actually know what you're doing in performing the conversion. The CLR will still raise a runtime error if 'il' does not reference a Customer object (the CLR will never let you assign an object reference to an incorrect Object type). You may wonder, if the CLR is going to perform a runtime check anyway, why require an explicit conversion? After all, VB6 is smart enough to do these conversions for you. This is actually a great new feature of VB.NET called Strict Type Checking. You can turn it off and have VB.NET perform these conversions for you without having to explicitly signal a conversion. However, this will be one of the many places in this book where I encourage you *not* to do so. Strict Type Checking is a wonderful new feature in VB.NET. It will improve your code. It will reduce bugs. And it will reduce the cost of bugs by helping you to find them more quickly. You should *turn on* Strict Type Checking as soon as you create any VB.NET project.5

Since the Customer object implements the ILinkList interface, all of the methods of that interface can be made directly available to the Customer object. The old rules of COM are not applicable. Because the Customer object is a superset of the Object methods and properties and any implemented interfaces, it is able to expose all of its methods as well as those of the implemented interfaces.

The remaining functions in the form should be easy to follow. The code is again just like that of the VB6 example except for the syntax changes you've seen so far. There is also a new cmdGC Command button that performs a garbage collection. Try clicking on this button

5 Because Microsoft blew it by leaving it off by default.

after removing an object from the list. You will see that it is deleted proving once again that VB.NET eliminates the circular reference problem:

```
    Protected Sub cmdRemove_Click(ByVal sender As System.Object, _

    ByVal e As System.EventArgs) Handles cmdRemove.Click
        Dim currentcustomer As Customer

        currentcustomer = CType(m_List, Customer)

        Do While Not currentcustomer Is Nothing
            If currentcustomer.CustomerName = CStr(lstCustomers().SelectedItem) Then
                currentcustomer.Remove(m_List)
                UpdateList()
                Exit Sub
            End If
            currentcustomer = CType(currentcustomer.NextItem, Customer)
        Loop
        UpdateList()

    End Sub

    Protected Sub cmdAdd_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdAdd.Click
        Dim newEntry As New Customer()
        newEntry.CustomerName = txtCustomerName().Text
        newEntry.Append(m_List)
        UpdateList()
    End Sub
    ' Click the GC button to force a garbage collection and see that
    ' abandoned objects are destroyed despite the circular reference.
    Protected Sub cmdGC_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdGC.Click
        gc.Collect()
        gc.WaitForPendingFinalizers()
    End Sub

End Class
```

Let’s take one last look at this example:

- VB.NET eliminates the circular reference problem present in the VB6 solution.

- By allowing direct access to all of the methods and properties of an object (including those on implemented interfaces), VB.NET dramatically simplifies coding when containment is used.

- The coding required in the Customer object to implement containment remains minimal.

In short, even without inheritance, VB.NET significantly improves the ease of code reuse through containment.

An Inheritance-Based Linked List Using VB.NET

In a sense, you have already seen inheritance. The Implements statement performs what is called interface inheritance—which exists in VB6 as well. Interface inheritance means that an object must implement all methods of an interface defined elsewhere (in an Interface definition in VB.NET or in a Class definition in VB6). With interface inheritance, an object can be referenced by a variable defined with the Interface type as well as one defined by the type of the object itself.

The LinkListNetInh project demonstrates the kind of inheritance everyone has been talking about for so long—true implementation inheritance.6 With implementation inheritance, an object inherits the actual implementation of its base class. The object can override that implementation if you wish. Also, with implementation inheritance, an object can be referenced by a variable defined with the base class type as well as one defined by the type of the object itself (the derived type).

In Listing 5-5, only the Customer object is changed as shown here in the Customer.vb file.

Listing 5-5. The LinkListNetInh Customer object.

```
' LinkList .Net example using inheritance
' Copyright © 2001 by Desaware Inc. All Rights Reserved
Public Class Customer
   Inherits LinkList

   Public CustomerName As String

   Public Sub New()
```

6 You may have heard of something called visual inheritance as well—perhaps in some .NET marketing material. There is no such thing. What they call visual inheritance is just regular implementation inheritance applied to an object like a form or a control that has visual characteristics (for example, it can be displayed or printed, has a user interface, etc.).

```
        MyBase.New()
        Me.Container = Me
    End Sub

    Protected Overrides Sub Finalize()
        system.diagnostics.Debug.WriteLine("Terminating customer " + CustomerName)
    End Sub
    ' There is no need to create a contained "LinkList" object -
    ' The customer object is a LinkList object as well.
    ' There is no need to implement the ILinkList interface, it's
    ' methods and properties are already implemented by the base class.
End Class
```

The private LinkList object is gone. So are all of the implemented functions. When the Customer object inherits the LinkList object, it becomes a LinkList object. We could, in fact, go back and modify the LinkList object so that it no longer used a Container variable—the LinkList object and the Customer object are now one and the same.

Pretty cool, isn't it?

Well, actually, it isn't.

Yes, this code works. Yes, it saves you a few lines of code, handling the aggregated object. But it doesn't change the client at all—and it doesn't make the Customer object any easier to use.

What really makes this code terrible is not so much technical as it is architectural. And it can be phrased thus:

> A customer is many things. A customer may be a person. A customer may be a corporation. A customer may be a government.
>
> But a customer is never a linked list.

Inheritance should only be used when you have a clear relationship in which the **inheriting object** *is an* **Inherited object**.

If this relationship does not exist, you should use interface inheritance and containment. The few extra lines of code to make this work are a small price to pay for the clarity of design (not to mention reduced support costs over the long term).

Curiously enough, this type of relationship is not very common in applications. It is common when building application frameworks that developers will use—thus (as you will soon see), the .NET framework itself uses inheritance extensively. You will be inheriting objects from the framework in every application and component you write. But you will rarely create objects intended to be inherited in turn.

The good news is that the elimination of the circular reference problem combined with the ease by which you can access methods of inherited interfaces makes containment an easy and effective way to reuse code.

A Dual Linked List Example

Here's a practical reason why inheritance is the wrong choice in this example. Linked lists are often used to place objects in order. What if you want to keep objects sorted two different ways at once? You might want the object to be present in two linked lists simultaneously. You can't inherit the same interface twice. You could define a new interface designed to support dual linked lists but you'll usually be better off with the approach shown in the LinkListNetDual project.

Listing 5-6 shows the revised LinkList class. It is similar to the previous one except that the container is a Public property. The architecture used here is also different from the previous example in that it follows the structure shown in Figure 5-1—the links reference the LinkList object itself rather than the Container object. The Container property can be used to navigate out to the actual object.

Listing 5-6. LinkList object revised to support dual use by a single object.

```
' LinkList .Net example showing dual lists
' Copyright ©2001 by Desaware Inc.  All Rights Reserved
Public Interface ILinkList

    ' This version is designed to link to the internal LinkList
    ' objects, using the public Container property to find the container
    Property NextItem() As ilinklist

    ' In this version all nodes are LinkList objects, so
    ' we need a public Container property to access the
    ' actual object
    Property Container() As Object

    ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList

    Sub Remove(ByRef Root As ILinkList)

    Sub Append(ByRef Root As ILinkList)
End Interface

Public Class LinkList
    Implements ILinkList

    ' This version is designed to link to the internal LinkList
    ' objects, using the public Container property to find the container
    Private m_Next As ILinkList
```

```
' We need to be able to navigate to the container
Private m_Container As Object

Friend Property Container() As Object Implements ILinkList.Container
   Get
      Container = m_Container
   End Get
   Set(ByVal Value As Object)
      m_Container = Value
   End Set
End Property

' Next is easy - just the next link.
Public Property NextItem() As ILinkList Implements ILinkList.NextItem
   Get
      NextItem = m_Next
   End Get
   Set(ByVal Value As ILinkList)
      m_Next = Value
   End Set
End Property

' In a single linked list, Previous has to search forward from the root
Public ReadOnly Property PreviousItem(ByVal Root As ILinkList) As ILinkList _
Implements ILinkList.PreviousItem
   Get
      Dim currentitem As ILinkList
      If (Root Is Me) Or (Root Is Nothing) Then
         Exit Property
      End If
      currentitem = Root
      Do
         If currentitem.NextItem Is Me Then
            PreviousItem = currentitem
            Exit Property
         Else
            currentitem = currentitem.NextItem
         End If
      Loop While Not currentitem Is Nothing
   End Get
End Property
```

```
    ' Remove has to search from root to find the previous node.
    ' Root must be by reference so it can be cleared if this
    ' is the last object, or reset if the first object
    Public Sub Remove(ByRef Root As ILinkList) Implements ILinkList.Remove
        Dim previtem As ILinkList

        previtem = PreviousItem(Root)
        If previtem Is Nothing Then
            Root = m_Next
        Else
            previtem.NextItem = m_Next
        End If
    End Sub

    ' Append searches from Root to the end of the list.
    Public Sub Append(ByRef Root As ILinkList) Implements ILinkList.Append
        Dim currentitem As ILinkList
        currentitem = Root
        If Root Is Nothing Then
            Root = Me
        Else
            While Not currentitem.NextItem Is Nothing
                currentitem = currentitem.NextItem
            End While
            currentitem.NextItem = Me
        End If
    End Sub
End Class
```

The Customer object shown in Listing 5-7 is considerably revised since it now supports linking the object into two lists at once. The object no longer inherits the ILinkList interface—an approach that limits you to a single linked list. Instead, it exposes its own methods for each list—for example, NextItem1 and NextItem2.

This approach requires more code to deal with the fact that the LinkList object only knows how to link objects that implement the ILinkList interface (like itself). The methods provided by the Customer object are references to other Customer objects.

Listing 5-7. The Customer object revised for use with multiple linked lists.

```
' LinkList .Net example showing dual lists
' Copyright ©2001 by Desaware Inc.  All Rights Reserved
Public Class Customer
```

```
Public CustomerName As String

' This version shows how a node can be in two lists at once.
' Note that the object does NOT Implement the LinkList interface
' Even though the links are done internally between LinkList objects,
' the outside world sees only Customer objects - thus all parameters
' and return values on link methods are Customer and not LinkList
Private m_MyLinkList1 As ILinkList
Private m_MyLinkList2 As ILinkList

Public Sub New()
   MyBase.New()
   m_MyLinkList1 = New LinkList()
   m_MyLinkList1.Container = Me
   m_MyLinkList2 = New LinkList()
   m_MyLinkList2.Container = Me
End Sub

Protected Overrides Sub Finalize()
   system.diagnostics.Debug.WriteLine("Terminating customer " + CustomerName)
End Sub

' Because we link into the contained object, we need
' a way to get access to the contained object in other nodes
Friend ReadOnly Property LinkList1() As ILinkList
   Get
      LinkList1 = m_MyLinkList1
   End Get
End Property

Friend ReadOnly Property LinkList2() As ILinkList
   Get
      LinkList2 = m_MyLinkList2
   End Get
End Property


' Functions require a bit more work to detect
' boundary conditions such as an empty list.
Public Sub Append1(ByRef Root As Customer)
   If root Is Nothing Then
      root = Me
   Else
```

```
        ' This line would fail if Root is Nothing
        m_MyLinkList1.Append(Root.m_MyLinkList1)
    End If
End Sub

Public Sub Append2(ByRef Root As Customer)
    If root Is Nothing Then
        root = Me
    Else
        m_MyLinkList2.Append(Root.m_MyLinkList2)
    End If
End Sub

' Again note how the implementation uses the ILinkList interface,
' but people using the Customer object only see references to
' Customer objects
Public ReadOnly Property NextItem1() As Customer
    Get
        Dim nextref As ILinkList
        nextref = m_MyLinkList1.NextItem
        ' We have to check the Nothing condition explicitly,
        ' otherwise the call to nextref.Container will fail.
        If nextref Is Nothing Then
            nextitem1 = Nothing
        Else
            ' nextref.container is of type Object. We need to convert explicitly
            nextitem1 = CType(nextref.container, Customer)
        End If

    End Get
End Property

Public ReadOnly Property NextItem2() As Customer
    Get
        Dim nextref As ILinkList
        nextref = m_MyLinkList2.NextItem
        If nextref Is Nothing Then
            nextitem2 = Nothing
        Else
            nextitem2 = CType(nextref.container, Customer)
        End If
    End Get
End Property
```

```
    Public ReadOnly Property PreviousItem1(ByVal Root As Customer) As Customer
        Get
            PreviousItem1 = CType(m_MyLinkList1.PreviousItem(Root.LinkList1), _
            Customer)
        End Get
    End Property

    Public ReadOnly Property PreviousItem2(ByVal Root As Customer) As Customer
        Get
            PreviousItem2 = CType(m_MyLinkList2.PreviousItem(Root.LinkList2), _
            Customer)
        End Get
    End Property


    Sub Remove1(ByRef Root As Customer)
        Dim llroot As ILinkList
        llroot = Root.LinkList1
        ' Why not just use m_MyLinkList.Remove Root.LinkList1?
        ' Because Root.LinkList1 will be placed in a temporary variable
        ' which is then called by reference. Changes to that temporary
        ' variable will not be magically reflected back to the Root.LinkList1 reference
        ' So we need to use our own temporary variable so that we can
        ' detect changes to that variable on this ByRef call.
        m_MyLinkList1.Remove(llroot)
        If llroot Is Nothing Then
            Root = Nothing
        Else
            Root = CType(llroot.Container, Customer)
        End If
    End Sub

    Sub Remove2(ByRef Root As Customer)
        Dim llroot As ILinkList
        llroot = Root.LinkList2
        m_MyLinkList2.Remove(llroot)
        If llroot Is Nothing Then
            Root = Nothing
        Else
            Root = CType(llroot.Container, Customer)
        End If
    End Sub
End Class
```

Listing 5-8 shows the code for the revised TestForm.vb form. This form adds a second listbox that contains only those objects whose customer name starts with letters before N. Note that the two variables that reference the lists, m_List and m_ListAtoM, are now both Customer types. The form is, in fact, completely oblivious to the ILinkList interface, which can't be used to access the Customer object anyway now that it is no longer inherited. The code for the form is otherwise similar to what you've seen before except that it has been extended to work with two lists.

Listing 5-8. The TestForm.vb form supports two linked lists.

```
' LinkList .Net example showing dual lists
' Copyright ©2001 by Desaware Inc.  All Rights Reserved

Public Class Form1
   Inherits System.Windows.Forms.Form

' Note the list roots are now Customers, not LinkList objects
   Private m_List As Customer
   Private m_ListAtoM As Customer


   ' Remove from both lists
   Protected Sub cmdRemove_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles cmdRemove.Click
      Dim currentcustomer As Customer

      currentcustomer = m_List
      Do While Not currentcustomer Is Nothing
         If currentcustomer.CustomerName = CStr(lstCustomers().SelectedItem) Then
            currentcustomer.Remove1(m_List)
            Exit Do
         End If
         currentcustomer = currentcustomer.NextItem1
      Loop
      currentcustomer = m_ListAtoM
      Do While Not currentcustomer Is Nothing
         If currentcustomer.CustomerName = CStr(lstCustomers().SelectedItem) Then
            currentcustomer.Remove2(m_ListAtoM)
            Exit Do
         End If
         currentcustomer = currentcustomer.NextItem2
      Loop
```

```
    UpdateList()

End Sub

' In this simple example, every customer with a name >"M" is excluded
' from the second list
' Note also how there is no need to use separate Customer
' and LinkList variables - we're always using the Customer interface
' only.
Protected Sub cmdAdd_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles cmdAdd.Click
    Dim newEntry As New Customer()
    newEntry.CustomerName = txtCustomerName().Text
    newEntry.Append1(m_List)
    If UCase(strings.Left(newEntry.CustomerName, 1)) <= "M" Then
        newEntry.Append2(m_ListAtoM)
    End If

    UpdateList()
End Sub

' Display the contents of both linked lists.
Private Sub UpdateList()
    lstCustomers().Items.Clear()
    lstAtoM().Items.Clear()

    Dim currentcustomer As Customer
    currentcustomer = m_List
    Do While Not currentcustomer Is Nothing
        lstCustomers().Items.Add(currentcustomer.CustomerName)
        currentcustomer = currentcustomer.NextItem1
    Loop

    currentcustomer = m_ListAtoM
    Do While Not currentcustomer Is Nothing
        lstAtoM().Items.Add(currentcustomer.CustomerName)
        currentcustomer = currentcustomer.NextItem2
    Loop

End Sub

Protected Sub cmdGC_Click(ByVal sender As System.Object, _
```

```
    ByVal e As System.EventArgs) Handles cmdGC.Click
        gc.Collect()
        gc.WaitForPendingFinalizers()
    End Sub

End Class
```

The LinkListVB6-2 sample project shows how you can implement a dual link list using VB6 (except that it still suffers from the circular memory reference problem).