Print ListView and TreeView Controls' Contents

Source: http://www.vb2themax.com/HtmlDoc.asp?Table=Articles&ID=50

by Marco Losavio

Note: As of April 2004, this article can no longer be found on vb2themax.com. It appears that they now only cover VB.NET and C#. I've decided to post this article to my Web site, http://CVibes.net, since I think there are still some Visual Basic 6 users who will find this information useful.

These two controls are frequently used to display structured data, so sooner or later you'll want to print their contents exactly as it appears to the end user.

With Visual Basic 5 came the capability to build ActiveX DLLs and OCXes, and since then a lot of VB developers have exploited it at their best. It is no coincidence that there are plenty of new VB-authored components in the commercial and shareware market, and in my opinion the reason is that VBers have always been users of such components, and therefore have very clear ideas about how they should be designed and implemented.

Whenever I have to create a new ActiveX control or a hierarchy of objects in an ActiveX DLL, I adopt my own, personal strategy. Instead of rushing to the code editor to implement the component, I start from the client side, and I try to imagine how a client application might use the control or the DLL. This gives me a good starting point for defining all the properties, methods, and events that the component should expose.

Recently, I had to repeatedly solve a recurring programming problem. In the company I work with we often use the ListView and TreeView controls as a means for displaying database data. These controls serve this purpose nicely, also because they let the end user adjust how data is display, for example by changing the width of a ListView's column, or expanding and collapsing nodes in a TreeView. The problem I had to face is: How can users print the contents of such controls exactly as it appears on their monitors? It would be beautiful if there were a method such as:

```
ListView1.Print
```

Unfortunately, no such method exists, so I had to roll up our sleeves and write all the code by myself. Faithful to my usual habits, I decided to start by defining what the above Print method is supposed to do, and write the actual code only later.

The Class Hierarchy

Firstly, it would be great if you could set a title with its font and alignment, and reserve some space on the printed page for the header and for the footer, for example to insert the page number. Even better, it should be possible to set page margins, the page size and orientation. I could go on with this requirement list, but I think that's enough for now. In order to easily set such a great number of properties, using a single object might quickly prove an inadequate approach. A much better solution is build a class hierarchy.

The root object in this hierarchy is the **CPrintListView** class, which exposes a Control property (to which you assign the ListView control to be printed) and the ImageList property (which receives a reference to a companion control that holds all the necessary icons):

```
Dim oSLV As New CPrintListView
Set oSLV.Control = ListView1
Set oSLV.ImageList = ImageList1
```

All page attributes and margins can be set through the secondary Page object:

```
With With oSLV.Page
   .BottomMargin = 2 'cm
  .TopMargin = 2
  .LeftMargin = 2
  .RightMargin = 2
  .Footer.Text = "Page. <pag>"
  .Footer.Font.Name = "Arial"
  .Footer.Font.Size = 8
  .Footer.Alignment = vbCenter
End With
```

while title attributes are defined through the Title object:

```
With oSLV.Title
   .Font.Bold = True
   .Alignment = vbCenter
   .Font.Italic = True
   .Font.Name = "Times New Roman"
   .Font.Size = 14
End With
```

Notice that the above code snippets make use of some VB types and objects, such as for the Font property (*StdFont*) and for the Alignment property (*AlignmentConstants*). This lets us edit code more quickly, thanks to IntelliSense.

The FontItem and HeaderFont properties are initially assigned a *copy* of the Font object exposed by the ListView control, so that you can later modify their individual font properties – such as Name, Size, Bold, Italic, etc. – without affecting the control's Font.

The hierarchy includes two more objects: **CPage** and **CText**. The former is used to define the page attributes and the latterm to define title, header and footer text. The CPrintListView e CPrintTreeView classes have a CPage property to define the page margins, orientation and size. Headers and footers are, in turn, defined as properties of CText type of the CPage object. The CText object allows to define a textual caption with font (*StdFont*) and alignment (*AlignmentConstants*) attributes. The complete class hierarchy is shown in Figure 1.

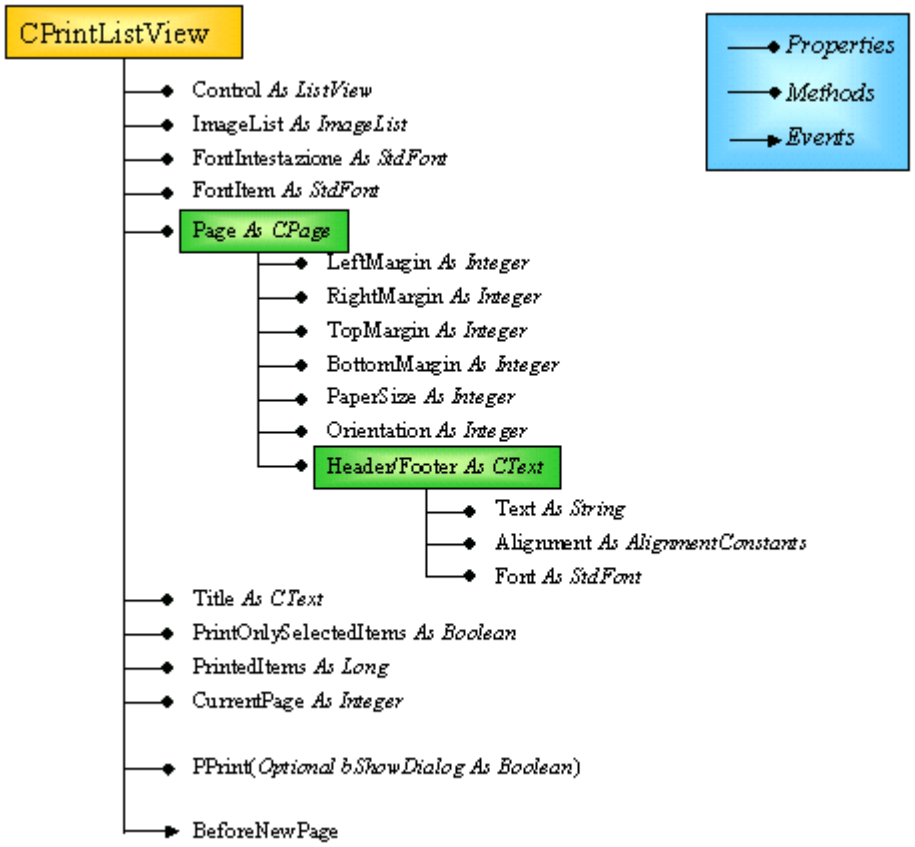

Figure 1A. *The CPrintListView object hierarchy.*

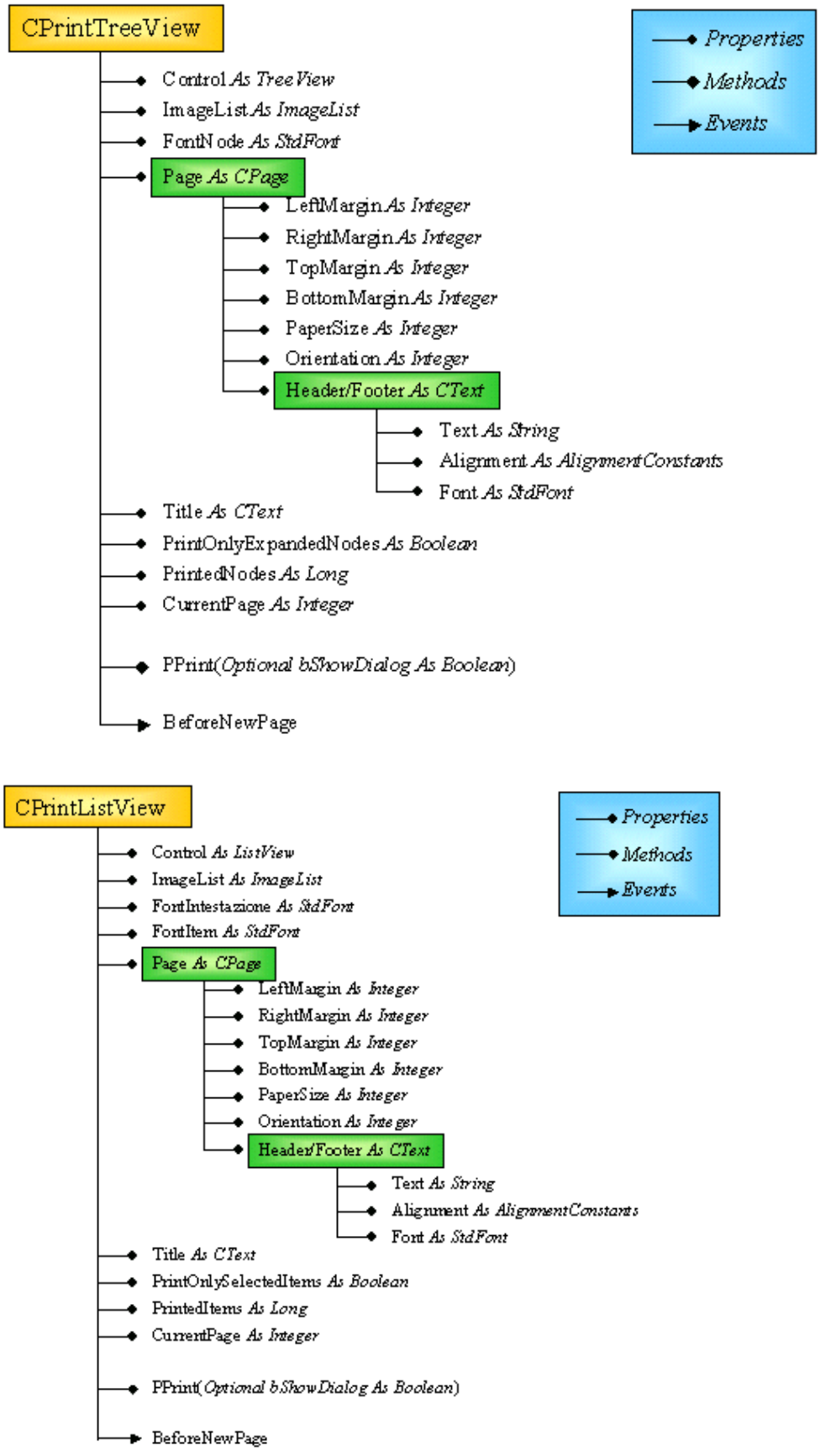

Figure 1B. *The CPrintTreeView object hierarchy.*

Printing a ListView control

The CPrintListView class can only print the contents of a ListView control in *lvwReport* view mode. In this view mode the ListView control has one or more columns, each one with its header, size and alignment. Moreover, every item is associated to an image which, together with the font, defines the overall printing row height. The CPrintListView class lets you define an header font (*HeaderFont*) and an item font (*FontItem*), which are initially set equal to the control's Font property. To print the ListView header the code iterates on the ColumnHeaders collection to read all column properties, as howed in Listing 1.

Listing 1. *ListView headers are printed by a loop on the ColumnHeaders collection. At each iteration the code checks the current column alignment in order to calculate where the print operation should start.*

```
Sub PrintHeaderLV(LV As ListView, lLeft As Long, lRight As Long)
  Dim ch As ColumnHeader
  Dim iCHWidth As Long
  Dim yMarg As Long

  yMarg = Printer.CurrentY

  For Each ch In LV.ColumnHeaders
    If ch.Width  0 Then
      Printer.Line (lLeft + iCHWidth, yMarg + _
        Printer.TextHeight("A")) - (lLeft + iCHWidth + _
        ch.Width, yMarg + Printer.TextHeight("A"))

      If ch.Alignment = lvwColumnLeft Then
        Printer.CurrentX = lLeft + iCHWidth
      ElseIf ch.Alignment = lvwColumnCenter Then
        Printer.CurrentX = lLeft + iCHWidth + (ch.Width _
          - Printer.TextWidth(ch.Text)) / 2
      ElseIf ch.Alignment = lvwColumnRight Then
        Printer.CurrentX = lLeft + iCHWidth + ch.Width _
          - Printer.TextWidth(ch.Text)
      End If

      Printer.CurrentY = yMarg
      Printer.Print ch.Text
      iCHWidth = iCHWidth + ch.Width
    End If
  Next
  Printer.CurrentY = Printer.CurrentY + 2 * Screen.TwipsPerPixelY
End Sub
```

Starting from the left margin (*Page.LeftMargin*) and setting the top margin (*yMarg*) the class calculates the CurrentX value by adding the width of each column:

```
For Each ch In LV.ColumnHeaders
    Printer.CurrentX = Page.LeftMArgin + iCHWidth
    Printer.CurrentY = yMarg
    Printer.Print ch.Text
    iCHWidth = iCHWidth + ch.Width
Next
```

To print the header with its correct alignment, the actual X coordinate where to print each column header is calculated as follows:

```
lLeft = Page.LeftMArgin
If ch.Alignment = lvwColumnLeft Then
    Printer.CurrentX = lLeft + iCHWidth
ElseIf ch.Alignment = lvwColumnCenter Then
    Printer.CurrentX = lLeft + iCHWidth + (cw - iTextWidth) / 2
ElseIf ch.Alignment = lvwColumnRight Then
    Printer.CurrentX = lLeft + iCHWidth + cw - iTextWidth
End If
```

where **cw** is the current column size to print (*ch.Width*) and **iTextWidth** is the text size (*ch.Text*) returned by the **TextWidth** Printer object's method.

To complete the header the code prints a separating line. To do so, at each cycle iteration, it prints a horizontal line whose length is equal to the current column width:

```
Printer.Line (lLeft + iCHWidth, yMarg + Printer.TextHeight("A"))- _
    (lLeft + iCHWidth + ch.Width, yMarg + Printer.TextHeight("A"))
```

To calculate the Y value the routine adds the top margin to the value that the Printer object's **TextHeight** method returns when the 'A' char is passed as an argument.

Printing individual cells of the controls is surely more complex than just printing the column headers. For one, we have to take page breaks into account, and decide when an item should be printed on the next page rather than on the current page. To do this we have to verify that the item height (*itmX.Height*) isn't greater than the remaining space on the page:

```
CanPrintItem = (Printer.CurrentY + hRow) < _
  (Printer.ScaleHeight - iBottomMargin)
```

Just before starting a new page, we must print the footer and raise an event that the programmer can trap to display the current printing page or to add dynamic data (Listing 2).

Listing 2. *The page footer is printed just before terminating the current page. Here the routine might be improved by raising an event (BeforeNewPage) in its client application.*

```
Function PrintControl(Optional ShowDialog As Boolean) As Boolean
  Dim itmX As ListItem

  On Error GoTo ErrStampaLV
  SetPage m_Page
  Printer.Print " "
  Printer.CurrentY = 0
  PrintCText m_Page.Header, m_Page.LeftMargin * vbCM, _
    m_Page.RightMargin * vbCM
  Printer.CurrentX = m_Page.LeftMargin * vbCM
  Printer.CurrentY = m_Page.TopMargin * vbCM

  PrintCText m_Title, m_Page.LeftMargin * vbCM, _
    m_Page.RightMargin * vbCM
  Printer.CurrentY = Printer.CurrentY + vbCM / 2
  SetPrinterFont m_FontHeader
  PrintHeaderLV m_Control, m_Page.LeftMargin * vbCM, _
    m_Page.RightMargin * vbCM
  SetPrinterFont m_FontItem
```

```
  m_PrintedItems = 0

  For Each itmX In m_Control.ListItems
    If (m_PrintOnlySelectedItems And itmX.Selected) Or _
      (Not m_PrintOnlySelectedItems) Then
      If Not CanPrintItem(itmX.Text, _
        m_ImageList.ImageHeight * Screen.TwipsPerPixelY, _
        m_Page.BottomMargin * vbCM) Then

        RaiseEvent BeforeNewPage
        Printer.CurrentY = Printer.CurrentY + _
          GetMax(Printer.TextHeight(itmX.Text), _
          m_ImageList.ImageHeight * Screen.TwipsPerPixelY)
        PrintCText m_Page.Footer, m_Page.LeftMargin * vbCM, _
          m_Page.RightMargin * vbCM

        Printer.NewPage
        PrintCText m_Page.Header, m_Page.LeftMargin * vbCM, _
          m_Page.RightMargin * vbCM
        Printer.CurrentY = m_Page.TopMargin * vbCM
        SetPrinterFont m_FontIntestazione
        PrintHeaderLV m_Control, m_Page.LeftMargin * vbCM, _
          m_Page.RightMargin * vbCM
        SetPrinterFont m_FontItem
      End If

      PrintItemLV itmX, m_Control, m_ImageList, _
        m_Page.LeftMargin * vbCM, m_Page.RightMargin * vbCM
        m_PrintedItems = m_PrintedItems + 1
    End If
  Next

  Printer.CurrentY = Printer.ScaleHeight - m_Page.BottomMargin
  PrintCText m_Page.Footer, m_Page.LeftMargin * vbCM, _
    m_Page.RightMargin * vbCM

  Printer.EndDoc
  PPrint = True
  Exit Function

ErrPrintLV:
  Printer.KillDoc
  MsgBox Err.Number & " " & Err.Description, vbExclamation,

"VB2THEMAX"

End Function
```

The printing of the icon that is associated to each row can be performed using the useful **PaintPicture** Printer object's method:

```
Printer.PaintPicture ilLV.ListImages(itmX.SmallIcon).Picture, iLeft, iTop
```

To print the text of the item and its subitems we can use the same approach used to print the header. To correctly calculate the Y coordinate we have to evaluate the height of each item (itmX.Height), which is the max between the image's height and the text's height plus one pixel. Because of this, the Y coordinate must be calculated so that the item is printed vertically with respect to the item's height (*hRow*):

```
yImage = Printer.CurrentY + (hRow – hImage) / 2
yText = Printer.CurrentY + (hRow – hText) / 2
```

Note that the image height can be obtained from the **ImageHeight** property of the ImageList control, whereas we must use the Printer object's **TextHeight** method to determine the height of the text.

TreeView Printing

At a first sight, printing the contents of a TreeView control might seem easier than printing a ListView control, because there are no headers and no columns, and the text is always left aligned. The truth, however, is that printing the TreeView nodes is probably *much* more complex than you probably imagine.

First, we have to determine the exact nodes sequence. This sequence can't be obtained by a mere cycle on the Nodes collection, because the nodes in this collection are in the order with which they have been added, and this can be different of the real nodes sequence displayed. The only way to retrieve the correct display order is to begin from the root node and get each next node by the **Child** and **Next** properties. The Child property returns the first child node, whereas the Next property returns the adjacent brother. These two properties must be used to call recursively the printing function. At the first iteration the printing function receives the root node:

```
Private Sub PrintANode(nodX As Node)
  'printing node code
  If nodX.Children  0 Then
    PrintANode nodX.Child
  End If
  If Not nodX.Next Is Nothing Then
    PrintANode nodX.Next
  End If
End Sub
```

Let's see now how we can print an individual Node object. First, we have to check whether the current node can be printed into the current page:

```
If Not CanPrintNode(nodX.Text, hImage, iBottomMargin) Then
  'Print the footer and got to new page
End If
```

The **CanPrintNode** function is a bit different from the CanPrintItem included in the CPrintListView class, because the Node object hasn't the Height property. To obtain a Node's height the routine calculates the max between the image height (*ImageList.ImageHeight*) and the text height returned by the Printer object's **TextHeight** method.

Next, we have to calculate the node's indentation level, by multiplying the depth level with the **Indentation** property. A simple method to evaluate the depth level is by counting the number of separation chars (*PathSeparator*) embedded into the **FullPath** property. This approach fails, however, if a Node's text can include the path separator. For instance, the default PathSeparator property value is '\' and the FullPath property for a node at second level is something like:

```
ParentText\CurrentNodeText
```

but if the text of the current node is 'Word1\Word2' then the FullPath property returns

```
ParentText\Word1\Word2
```

In this case, just counting the number of separator characters we'd calculate an incorrect depth level. We can work around this problem by using a path separator character that we know for sure never appears in Nodes' Text property or, better yet, we can use a cycle that counts the number of parents:

```
iLevel = 0
While Not nodX.Parent Is Nothing
  iLevel = iLevel + 1
  Set nodX = nodX.Parent
Wend
```

Multiplying the depth level value by the **Indentation** property we get the X coordinate value at which the node should print. The code that prints the node's text and image is similar to the one used for left-aligned ListView items. But wait! We are forgetting an important detail: the lines that link nodes with their parents! It turns out that the programming logic that ensures that all such lines are correctly printed isn't as easy as one could hope. To understand the algorithm on which the CPrintTreeView class is based you should have a look at the code in Listing 3.

Listing 3. *Printing the lines that connect TreeView nodes.*

```
'Printing the link lines

iCurrentY = Printer.CurrentY
If Not nodX.Parent Is Nothing Then
  'horitzontal line
  Printer.Line (m_Page.LeftMargin * vbCM + iLevel * iIndentation - _
    iIndentation / 2, iCurrentY + hRow / 2)- _
    (m_Page.LeftMargin * vbCM + iLevel * iIndentation - 2 * _
    Screen.TwipsPerPixelX, iCurrentY + hRow / 2)

    If nodX.Next Is Nothing Then
      Printer.Line (m_Page.LeftMargin * vbCM + iLevel * _
        iIndentation - iIndentation / 2, iCurrentY)- _
        (m_Page.LeftMargin * vbCM + iLevel * iIndentation - _
        iIndentation / 2, iCurrentY + hRow / 2)
    Else
      Printer.Line (m_Page.LeftMargin * vbCM + iLevel * _
        iIndentation - iIndentation / 2, iCurrentY)- _
        (m_Page.LeftMargin * vbCM + iLevel * iIndentation - _
        iIndentation / 2, iCurrentY + hRow)
    End If

    Set nodXX = nodX
    For j = iLevel To 1 Step -1
      Set nodP = nodXX.Parent
      If Not nodP Is Nothing Then
        If Not nodP.Next Is Nothing And Not nodP.Parent Is _
          Nothing Then
          Printer.Line (m_Page.LeftMargin * vbCM + _
            iIndentation / 2 + (j - 2) * _
            iIndentation, iCurrentY)- _
            (m_Page.LeftMargin * vbCM + iIndentation _
            / 2 + (j - 2) * iIndentation, _
            iCurrentY + hRow)
        End If
      End If
      Set nodXX = nodP
      If nodXX Is Nothing Then Exit For
    Next j
  End If
```

First of all, lines are printed only if the current Node has a Parent, and its parent is expanded (this routine doesn't print link lines for any root node):

```
If Not nodX.Parent Is Nothing Then
  'Print the lines
End If
```

Next, we print the horizontal line starting from the current node X coordinate until the half of the Indentation property value, so to get to the middle point of the Parent node's width:

```
Printer.Line (iCurrentX - iIndentation / 2, iCurrentY + hRow / 2) _
  - iCurrentX - 2 * Screen.TwipsPerPixelX, iCurrentY + hRow / 2)
```

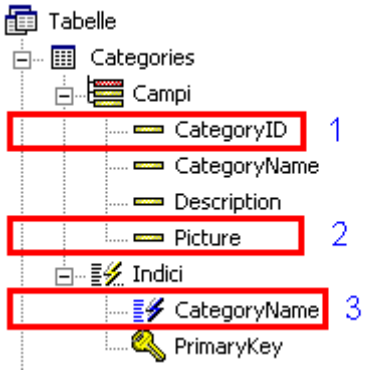

Figure 2. *These three cases relate to the tests to be performed when drawing connecting lines in a TreeView control.*

The height of the vertical line which is perpendicular to the horizontal line just printed depends on whether there is another brother node (cases 1 and 2 of Figure 2):

```
If nodX.Next Is Nothing Then
  Printer.Line (iCurrentX - iIndentation / 2, iCurrentY) _
    - (iCurrentX, iCurrentY + hRow / 2)
Else
  Printer.Line (iCurrentX - iIndentation / 2, iCurrentY) _
    - (iCurrentX, iCurrentY + hRow)
End If
```

The last step is the most difficult one: we need to print all lines that are correctly linked to the expanded node lines up in the hierarchy. From Figure 2 we note that the vertical line printing has to be printed if a parent exists and it has a brother (case 3 in Figure 3). This step must be repeated for all the levels except for the very first, which contains the root nodes:

```
For j = iLevel To 2 Step –1
  Set nodP = nodP.Parent
  If Not nodP Is Nothing Then
    If Not nodP.Next Is Nothing Then
      Printer.Line (iLeftMargin + iIndentation / 2 + (j - 2) _
        * iIndentation, iCurrentY)- (iLeftMargin + _
        iIndentation / 2 + (j - 2) * iIndentation, _
        iCurrentY + hRow)
    End If
  End If
Next i
```

What we've left out

The example enclosed isn't complete, however, and it should be considered as a starting point to extend and improve according to your own requirements. For example, the CPrintListView class could implement an automatic *wordwrap* mechanism when the string to print is longer than the correspondent column width. It could also support multiple header styles (square, grid...) and the capability to print borders and grid lines. Before and after printing an item the class might raise an event (**BeforePrint** and **AfterPrint**) so to change the item values or to skip the printing of the current item providing the classic *Cancel* parameter. The **AfterPrint** event can be used to add graphic or additional data, such as secondary descriptions or totals and subtotals. On request, the class might be able to generate automatic totals on any numeric column, to be print at the end of each page or at the end of the report.

To give the programmer the ability to display a percent progress bar, the routine should raise an event (**ItemPrinted**) at the end of every item/node printing. Here the programmer could use **PrintedItems** and **PrintedNodes** properties to determine the exact printing percentage.

In the enclosed example the CPrintListView and CPrintTreeView classes use the **Page** object to set page margins, but they don't take into account the non-printable area of the page. The size of this area can be determined by calling the GetDeviceCaps API with PHYSICALOFFSETX/Y constant, and it should be subtracted from the page margin so the printing begins at the exact position. Another possible improvement is in the CPrintTreeView class, that doesn't support printing neither root nodes lines nor the plus/minus symbols beside each Node.

Finally, the **PrintControl** method has a boolean parameter (*ShowDialog*) which isn't currently used. The purpose of this is to display a Print common dialog to let the user select the printer and set other printing parameters, such as whether the routine should print selected items or expanded nodes only.

What about an automatic abort dialog? or a print preview? Any suggestion is welcome.

Conclusion

Doubtlessly, writing classes is the best way to learn how to augment the capabilities of an existing control. Unfortunately, there is one problem: we can't easily compile these classes into a separate ActiveX DLL. This problem occurs when an ActiveX control such as a ListView, a ImageList or a TreeView is exposed by a class property. Once the DLL is compiled, it won't correctly work on another computer. In fact, the instructions

```
Set oSLV.Control = ListView1
Set oSLV.ImageList = ImageList1
```

raise a "Type Mismatch" error message. This error occurs because these properties implicitly reference the Extender object silently created by VB for each ActiveX control loaded in the IDE; such Extender object is different from machine to machine, which explains why you can't compile the DLL on a system and use it on another one. In other words, you must always include these classes in your projects, which is only a minor inconvenience if compared with the capabilities that these classes add to your projects.
