Bonus Chapter 2

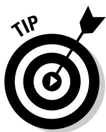
VBA Programming in Visio

In This Chapter

- Understanding how VBA can make Visio better
- Working with Visio-related objects
- Responding to Visio events

Visio helps you draw everything from floor plans to electronics circuits to organizational charts. In fact, it would be hard to imagine anything of a business nature that you can't draw with Visio, except pure graphics. Visio isn't an artist's tool, and you can't use it as a Computer Aided Drafting (CAD) replacement; it's the tool for the rest of us. I've used Visio for many years to draw the illustrations for my books and to accomplish personal tasks, such as creating woodworking designs. It's even possible to use Visio to design software.

Because you can draw such a huge array of things with Visio, it would be hard to describe everything you can do with it in a single chapter. However, the vast array of drawing templates makes it possible to use VBA with Visio in a big way, such as to automate common Visio drawing tasks. For example, you could use a macro to set up a drawing, complete with company logo and standardized title block, for particular customers.



A single chapter can't tell you about Visio itself in much detail, so I'm assuming that you know how to use the product to perform basic tasks. My book *Visio 2007 For Dummies* (published by Wiley), provides you with a step-by-step look at everything Visio can do and with only a little drawing ability on your part. You'll be amazed at how much you can do with Visio — everything from creating organizational charts to analyzing data for a report.

Using Visio with VBA

Visio has a significant following of VBA developers because they can do so much with Visio drawings. In fact, Microsoft has created a Visio Developer Portal (`http://msdn2.microsoft.com/en-us/library/aa395291.aspx`) in recognition of the level of developer participation. The following list provides you with some ideas of how you can use VBA to make Visio even better than it already is, but really, this is just the tip of the iceberg:

- Perform automatic shape settings and request user input for more.
- Automatically resize and reorganize the diagram as needed to accommodate new shapes.
- Create interactive diagrams for what-if analyses and public demonstrations.
- Let Visio interact with other Office applications, such as Word (you can see a floor plan solution at `http://msdn.microsoft.com/library/en-us/dnvisio00/html/executiveoffice.asp`).
- Analyze data and present the output in graphical form, such as the network costing solution shown at `http://msdn.microsoft.com/library/en-us/dnvisio00/html/executiveoffice.asp`.
- Interact with external data sources (see the Smart Card reader demonstration at `http://msdn.microsoft.com/library/en-us/dnvisio00/html/executiveoffice.asp`).
- Traverse connected diagrams to perform analysis on data relations — a sort of data mining (see the stock portfolio example at `http://msdn.microsoft.com/library/en-us/dnvisio00/html/executiveoffice.asp`).
- Create parts lists based on the data in one or more diagrams.
- Automatically update support documents, such as organizational charts, based on changes to a central database.
- Define new shape behaviors that can automate real-world tasks, such as parts inventory and ordering.

Understanding the Visio-Related Objects

Visio doesn't provide a global means of working with VBA. It doesn't include an application-level module like FrontPage or a Normal.dot (Normal.dotm for Word 2007) solution. However, you do have options with Visio that aren't apparent immediately. Generally, you start creating a VBA application for Visio by creating a drawing. However, after you have the drawing completed,

you can save it as a template. Every drawing you create from that template includes the macros you've created.

Macros you create in a template don't automatically update the associated diagram. Visio uses the template to create the diagram, but there isn't any permanent connection between the template and the diagram. Consequently, the macros that exist in a template at the time you create the diagram also exist in the diagram, but any changes you make to the template later don't appear in the diagram. You must update the diagram macros separately.

The Visio object model (`http://msdn.microsoft.com/library/en-us/vissdk11/html/viobjtocMain_HV01066071.asp`) shows that Visio provides access to the document, the shapes it contains, the user interface, windows, and the current selection — everything you would expect to access from VBA. Visio 2007 uses the standard toolbar-and-menu interface, so you'll find the usual `CommandBar` objects. One difference with Visio VBA programming is that many tasks revolve around user drawing activities, so you have to pay special attention to events (see the table at `http://msdn.microsoft.com/library/en-us/vissdk11/html/vievtEventCodes_HV81901708.asp`).

When working with Visio diagrams, you need to consider several objects that include `Document`, `Page`, `Layer`, `Shape`, and `Cell`. Each of these objects has a corresponding physical representation. A *document* is the entire diagram, and it usually resides in a single file. A document can have multiple pages. Each *page* represents a single drawing within the document, such as a single room within an office or a circuit board within a larger device. A page can have multiple layers. A *layer* provides a means of separating drawing elements into groups, such as movable furniture and electronics. Layers can also contain revision marks or any other drawing element that requires separation from other drawing elements. A page can contain one or more shapes. A *shape* represents something you want to draw, anything from a piece of furniture to a box. Even text and connectors are shapes within Visio. Finally, even though a user never actually sees any cells, Visio uses them to hold information about shapes. Think of a *cell* as an individual data element in a spreadsheet-like organization of data about individual shapes in the diagram.

Microsoft has added a number of new VBA objects and methods to Visio 2007. All these new features provide access to the new templates and stencils that Visio 2007 provides, along with the increased data access functionality. You can find a list of these objects and methods at `http://msdn2.microsoft.com/en-us/library/aa395291.aspx`.

Visio works with several libraries. In addition to the standard Office, VBA, and OLE (Object Linking and Embedding) Automation libraries, a minimal Visio setup also includes the Visio library. As with any other VBA application, you can add more libraries as needed. However, everything you need to interact with Visio itself appears as part of the Visio library. It's interesting to note

that Visio provides a considerable number of non-default libraries, as defined in the following list:

- Microsoft Visio 12.0 Diagram Launch Control
- Microsoft Visio 12.0 Drawing Control Type Library
- Microsoft Visio 12.0 Save As Web Type Library
- Microsoft Visio Database Modeling Engine Type Library
- Microsoft Visio UML Add-in for Microsoft Visual C++ 6.0
- Microsoft Visio UML Solution for Visual Basic Type Library

You can add any of these libraries to your application by choosing Tools➪References to display the References dialog box. Place a check mark next to each library you want to add and then click OK.

The diagram launch control library contains features for creating new diagrams. The drawing control library is interesting because it consists entirely of events. This library is the one to load when you don't find an event you need. For example, this library includes a `BeforeMasterDelete` event that fires whenever someone decides to remove a master page. The Save As Web (SAW) feature lets you output your diagrams as a Web page. The database modeling library is new with Visio 2007. It comes into play only when you interact with databases. For example, you might want to add shape data based on database input. Several of these libraries are specific to certain kinds of diagrams. For example, the Unified Modeling Language (UML) libraries are for software engineering diagrams.

Using the Application object

As with every other Office application, the `Application` object provides access to the rest of Visio. I purposely created this macro in a template so that you can see that templates really can contain macros. Any diagram you create using this template also contains the macros. Listing BC2-1 shows how you can interact with the `Application` object. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/vbafd5e`.)

Listing BC2-1 Using the Application Object

```
Sub UseApplication()
    ' Holds the description.
    Dim Description As String

    ' Get the document description.
    With Application.ActiveDocument
        Description = _
```

```
                "Name: " + .Name + vbCrLf + _
                "Description: " + .Description + vbCrLf + _
                "Paper Height: " + _
                CStr(.PaperHeight("inches")) + vbCrLf + _
                "Paper Width: " + _
                CStr(.PaperWidth("inches")) + vbCrLf + _
                "Paper Size: "

            ' The paper size requires special handling.
            Select Case .PaperSize
                Case VisPaperSizes.visPaperSizeA3
                    Description = Description + "A3"
                Case VisPaperSizes.visPaperSizeA4
                    Description = Description + "A4"
                Case VisPaperSizes.visPaperSizeA5
                    Description = Description + "A5"
                Case VisPaperSizes.visPaperSizeB4
                    Description = Description + "B4"
                Case VisPaperSizes.visPaperSizeB5
                    Description = Description + "B5"
                Case VisPaperSizes.visPaperSizeC
                    Description = Description + "C"
                Case VisPaperSizes.visPaperSizeD
                    Description = Description + "D"
                Case VisPaperSizes.visPaperSizeE
                    Description = Description + "E"
                Case VisPaperSizes.visPaperSizeFolio
                    Description = Description + "Folio"
                Case VisPaperSizes.visPaperSizeLegal
                    Description = Description + "Legal"
                Case VisPaperSizes.visPaperSizeLetter
                    Description = Description + "Letter"
                Case VisPaperSizes.visPaperSizeNote
                    Description = Description + "Note"
                Case VisPaperSizes.visPaperSizeUnknown
                    Description = Description + "Unknown"
            End Select
        End With

        ' Get the active page description.
        With Application.ActivePage
            Description = Description + vbCrLf + _
                "Page Width: " + _
                CStr(.Shapes("ThePage").Cells("PageWidth")) + _
                vbCrLf + "Page Height: " + _
                CStr(.Shapes("ThePage").Cells("PageHeight"))

        End With

        ' Display the description on screen.
        MsgBox Description, _
               vbInformation Or vbOKOnly, _
               "Document and Page Description"
    End Sub
```

The code begins by accessing the current document using the `ActiveDocument` object. You must remember that Visio has several drawing layers. The document can contain multiple pages. Each page can contain several layers. A page can also have a background page associated with it. All these issues are important as you work with VBA. The `ActiveDocument` object tells you only about the document, not about the pages it contains. Consequently, when you access the `PaperHeight` and `PaperWidth` properties, you're looking at the print specifications for the document, not the actual size of the pages.

Visio also has a few odd conventions that you might not have seen in the past. Notice that the `PaperHeight` and `PaperWidth` properties both have what appears to be an index of `inches`. However, this argument is merely a unit of measure. You'll find that Visio requires units of measure in many cases. Also, because this property is a `Double`, you must convert it to a `String` using the `CStr()` function.

You run into more than a few enumerations in Visio, and the `PaperSize` property shows just one of many cases. If you want a text equivalent for the enumeration, you must create a `Select Case` statement such as the one shown in the code.

Visio provides the `ActivePage` object so that you can interact with the currently selected page. Pages are all about shapes. In fact, the page itself is a shape, which is why you use `Shapes("ThePage")` to access the page. Within a particular shape are cells that contain various pieces of information. The `Cells("PageWidth")` contains the width of the page, and the `Cells("PageHeight")` contains the height of the page. Figure BC2-1 shows the output of this application. Notice that the page size is indeed different from the document size.

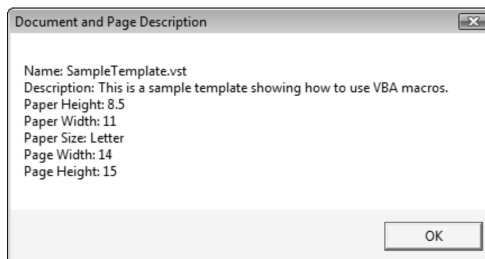
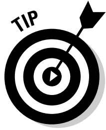


Figure BC2-1: Listing diagram information is just one use of VBA in Visio.



You might wonder where `ThePage` comes from as an index in the listing. It isn't a magic name that you just have to know in order to work with Visio. Sometimes the name of a shape isn't obvious and you want to determine it quickly without writing a lot of code. If you have a specific shape in mind, all you need to do is choose View➪Shape Data Window. The title bar of the Shape Data window always contains the name of the shape as you need it for use with VBA. This technique even works with what appears to be unnamed shapes. Select text, for example, and you see that it usually has the name Sheet.XX, where *XX* is a number.

Listing multiple Page objects

Many drawings require that you work with just a single page. However, it's likely that you'll eventually have to work with multiple pages in a single diagram. For example, an office layout might include a single page for each office. In short, you'll find a need, at some point, for viewing individual pages. Listing BC2-2 shows how to access each page and display its name. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/vbafd5e`.)

Listing BC2-2 Locating and Listing Multiple Drawing Pages

```
Sub ListPages()
    ' Holds the list of pages.
    Dim ThePages As Pages

    ' Holds the page information.
    Dim PageNames As String
    Dim ThisPage As Page

    ' Get the list of pages.
    Set ThePages = ActiveDocument.Pages

    ' Obtain the page information.
    For Each ThisPage In ThePages

        ' Check for a drawing page.
        PageNames = PageNames + ThisPage.Name +
            CStr(ThisPage.ObjectType) + vbCrLf
    Next

    ' Display the results.
    MsgBox PageNames, vbInformation Or vbOKOnly, "Drawing
           Pages in Document"
End Sub
```

The code for this part of the example looks like the code used to work through many collections in the book. It begins by accessing the `Pages` collection in the `ActiveDocument` object. The code retrieves an individual page from the collection and places its name in `PageNames`. Finally, the example displays the list of pages, as shown in Figure BC2-2.

Notice that the code in this example lists all the pages. You can limit the pages shown by detecting property values in the shapes' cells. The "Working with Shape cells" section of this chapter describes this concept in detail. The important information for this example is that documents contain multiple pages, in most cases, even if one of those pages is a background and doesn't contain any actual diagram data.

Listing the Shape objects in a drawing

The whole point of using Visio is to place shapes from a stencil onto a page. That's how you draw, and it's one of the reasons that Visio is so easy for non-artists to use. This example demonstrates how to access shapes within a page. Normally, you use the active page, as shown in Listing BC2-3, but you can also work with pages by using the `Pages` collection associated with any document. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/vbafd5e`.)

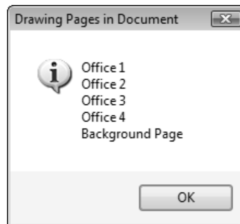


Figure BC2-2: Displaying all the pages in a document.

Listing BC2-3 Accessing the Shapes in a Drawing

```
Sub ListShapes()
    ' Holds the list of shapes for a page.
    Dim TheShapes As Shapes

    ' Obtain the list of shapes.
    Set TheShapes = ActivePage.Shapes

    ' Holds individual shape data.
    Dim ThisShape As Shape
```

```
    Dim ShapeNames As String

    ' Obtain each shape and add it to the list.
    For Each ThisShape In TheShapes
        ShapeNames = ShapeNames + ThisShape.Name + vbCrLf
    Next

    ' Display the results onscreen.
    MsgBox ShapeNames, _
           vbInformation Or vbOKOnly, _
           "Shapes on Current Page"
End Sub
```

The example works through the `Shapes` collection found in the `ActivePage` object. Notice that you don't have to even consider the document, in this case, because the `ActivePage` object brings the current page to the foreground. Because most of your Visio scripts will work with the shapes that the user is currently drawing, you'll find that using the `ActivePage` object is an extremely efficient way to write code. The output from this example appears in Figure BC2-3.

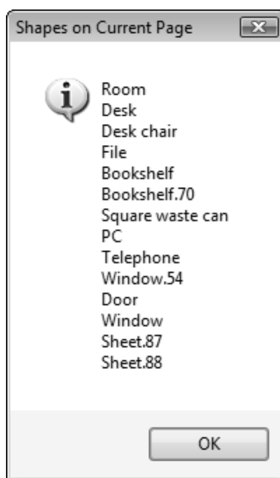


Figure BC2-3: Each shape on a page appears as part of that page's `Shapes` collection.

You should notice something about the listing of shapes in this dialog box. First, the first item of every shape type has that shape's name. For example, the first bookshelf has the name `Bookshelf`. All shapes after the first shape have a random number added to their name. In this case, the second bookshelf in the room has the name `Bookshelf.70`. You need to exercise care when writing shape code to consider this naming convention.

Second, all objects have a name, even objects that you might not think have a name. For example, some people might not think that a room will have a name, but it does, in this case: Room. In addition, text and connectors also have names. The Sheet.87 and Sheet.88 shapes in the listing are text added to the diagram. This text is independent of any shape. Shape text doesn't have a name because it's associated with a property within the shape itself. Consequently, when you see a shape name such as Sheet.87, you know that the diagram contains an independent, unnamed element, such as text or a connector.

Working with Shape cells

Shape cells can be a difficult concept to understand unless you think about them using some physical equivalent. My preference is to look at them as the cells within a spreadsheet. Visio refers to cells by the terms `Section`, `Row`, and `Column`. Think of a section as a page within the spreadsheet file. Just as you use pages to hold different categories of data, a *section* holds different categories of shape data. When you look at a spreadsheet, you see data organized by row and column. The rows and columns for a shape serve the same purpose. A *row* might hold an individual data property, such as the `Name` field for an organizational chart block. A *column* might hold a description of that property, such as the property's data type. The actual row and column definitions can vary by section, but the idea is always one of organizing the data in some way.

Much of the Visio code you see doesn't work with sections, rows, and columns, however, because Visio provides an easier method of locating a particular cell using an index. For example, a cell index might appear as `Prop.Title`. This index tells you that the cell is a property and that it's for the `Title` field. However, many of the cell index names aren't nearly so easy to figure out.

Unfortunately, as you view examples online and in the Visio help files, the cell indexes will prove elusive. You won't find a listing of them anywhere because the names can be literally anything you choose. Many people give up trying to figure them out in frustration. The focus of the example in this section is to provide you with a quick and easy method for discovering all the cell indexes associated with a given shape. Everything you need to perform this task appears in Listing BC2-4. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/vbafd5e`.)

Listing BC2-4 Getting to the Cell Level of Visio

```
Sub ListCells()
    ' Holds the current shape.
    Dim TheShape As Shape

    ' Loop counter variables.
    Dim RowCount As Integer
    Dim CellCount As Integer

    ' Holds the current cell information.
    Dim TheCell As Cell
    Dim CellName As String

    ' Obtain a selected shape.
    Set TheShape = ActivePage.Shapes("Telephone")

    ' Open the file that will contain the cell names.
    Open ThisDocument.Path + "\CellNames.txt" For Output
          As #1

    ' Process each of the cell rows.
    For RowCount = 0 To TheShape.RowCount(visSectionProp)
          - 1

        ' Process each cell in the row.
        For CellCount = 0 To
          TheShape.RowsCellCount(visSectionProp,
          RowCount) - 1

            ' Obtain the specific cell.
            Set TheCell =
          TheShape.CellsSRC(visSectionProp, RowCount,
          CellCount)

            ' Save the name of the Cell.
            CellName = TheCell.Name + vbCrLf

            ' Output the data.
            Write #1, CellName
        Next
    Next

    ' Close the file.
    Close #1

End Sub
```

The code begins by accessing a particular shape. You can determine the shape name by opening the Shape Data window, selecting the shape you want to work with, and viewing its name on the title bar. As an alternative, you can always use the `ListShapes` macro, shown in listing BC2-3.

After the code obtains a reference to the shape, it opens a text file on disk to store the information. Using this approach makes it easier to reference the information later. Figure BC2-4 shows typical output from this application. Each entry tells you about a particular shape property element. In this case, you're looking at the `ShapeClass` field, which includes all the usual elements, including a prompt, label, and format. I showed this particular field because you can't access it directly within Visio, but you can access it by using VBA code.

Now that the code has a file to use, it proceeds to scan through the rows and columns for a particular section. The section name you choose determines the kind of data you receive from Visio because Visio places each object data type in a different section. The `VisSectionIndices` enumeration contains a list of sections that Visio recognizes. However, you'll use some sections more than others. The `visSectionObject` section returns non-repeating standard object values, such as line color, height, and width. The `visSectionProp` section contains definitions of the properties you create for an object, including the values of those properties. The `visSectionCharacter` section stores all the font information for the text displayed in the shape.

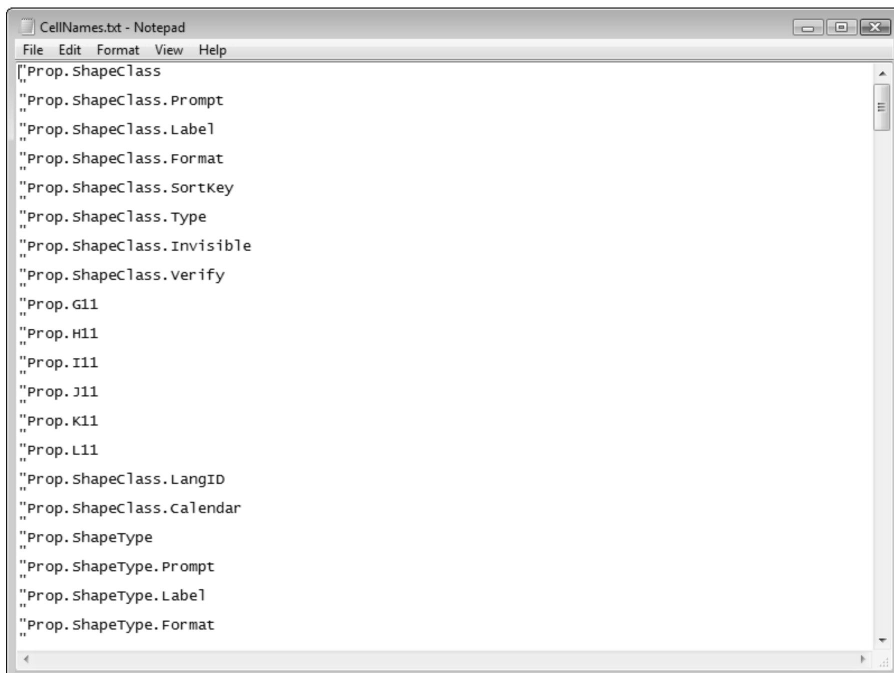


Figure BC2-4: The `Cell` index names can be a mystery until you discover how to list them.

Notice how the code relies on Visio to tell it about the shape data. Do not assume anything about the shape data because the data varies from shape to shape. The `TheShape.RowCount()` function always returns the number of rows for a particular section of the shape data. When you know the row you want to work with, you can check for the number of cells in that row by using the `TheShape.RowsCellCount()` function. The code obtains the name of the individual cells and writes them to disk at this point. After you find the name of the cell you want to work with, you can display its value using code like this:

```
TheShape.Cells("Prop.AssetNumber").ResultStr("inches")
```

In this case, the code actually returns the asset number. If you want to obtain one of the asset number property values, for example, such as the data type, you use `Prop.AssetNumber.Format` or another cell name instead. The point is that you can control every aspect of a property programmatically.

The code ends by closing the output file. Always make sure that you close the file. Doing so ensures that the file contains all the data and reduces the risk of introducing memory or resource leaks into your application.

Handling the ShapeAdded event

Visio includes a wealth of events. In fact, it may possibly provide more events than you'll ever interact with for any other Office application simply because many VBA applications for Visio seem to respond to the user's normal performance of a task. One event that you'll use most commonly is the `ShapeAdded` event. This event fires every time the user places a shape on a page.

You can add events to your code in a number of ways. However, the easiest method is to let Visio do the work for you. The following steps help you insert an event handler into your application:

1. **Double-click `ThisDocument` in the Project Explorer window.**

   You see a Code window appear. Visio opens a new Code window even if you already have one open. Use this Code window for all Visio event handlers.

2. **Choose Document in the Object field (on the left) of the Code window.**

3. **Choose an event, such as `ShapeAdded`, in the Procedure field (on the right) of the Code window.**

   Visio automatically creates a `Sub` for you. The `Sub` has any variables needed for the event handler defined for you. For example, choosing `ShapeAdded` creates an event handler with `ByVal Shape As IVShape` defined.

After you define a new event handler, you can add code to it to perform a task based on a user event. In this example, the code checks for particular shape types and reacts to them. Listing BC2-5 shows everything you need to begin this example. (You can find the source code for this example on the Dummies.com site at `http://www.dummies.com/go/vbafd5e`.)

Listing BC2-5 Handling Visio Events

```
Private Sub Document_ShapeAdded(ByVal Shape As IVShape)
    ' Create an object to determine the shape type.
    Dim ShapeType As Master
    Set ShapeType = Shape.Master

    ' Determine the shape type.
    If Left(ShapeType.Name, 9) = "Executive" Then
        ' Change the executive block to the owner's name
        ' and title.
        Shape.Cells("Prop.Name").Formula = _
            "=""George Smith"""
        Shape.Cells("Prop.Title").Formula = _
            "=""President"""

    ' Managers require special handling.
    ElseIf Left(Shape.Name, 7) = "Manager" Then

        ' Holds the user response.
        Dim Response As VbMsgBoxResult

        ' Ask the user about the person's title.
        Response = MsgBox("Add a Vice President?", _
            vbYesNo Or vbQuestion, _
            "Choose an Organization Chart Type")

        ' Create the block based on the user's input.
        If Response = vbYes Then

            ' Use the appropriate title.
            Shape.Cells("Prop.Title").Formula = _
                "=""Vice President"""

            ' Set the shape's line characteristics.
            Shape.Cells("LineColor").Formula = _
                "=THEMEGUARD(RGB(128,0,0))"
            Shape.Cells("LinePattern").Formula = _
                MsoLineDashStyle.msoLineDash
            Shape.Cells("LineWeight").Formula = "=1 pt."
        Else

            ' Use the appropriate title.
```

```
            Shape.Cells("Prop.Title").Formula = _
                "=""Department Head"""

            ' Set the shape's line characteristics.
            Shape.Cells("LineColor").Formula = _
                "=THEMEGUARD(RGB(0,128,0))"
            Shape.Cells("LinePattern").Formula = _
                MsoLineDashStyle.msoLineSquareDot
            Shape.Cells("LineWeight").Formula = "=0.8 pt."
        End If

    ' For everyone else, just display the organization
    ' chart type.
    Else
        MsgBox "You've added a " + ShapeType.Name, _
            vbOKOnly Or vbInformation, _
            "New Shape Added"
    End If
End Sub
```

The code begins by checking the shape type. You can determine the shape type in a number of ways. However, checking the shape name should work fine as long as you remember that the shape name can include a period and numbers. For example, an `Executive` shape always begins with the name `Executive`, even when the actual shape name is `Executive.22`. Consequently, the code relies on the `Left()` function to obtain the important part of the `Name` property string.

When the shape is an executive, the code changes the value of the `Name` and `Title` fields to match the executive's name and title. You can use this technique with any shape that has a consistent value in your organization. Using this approach saves time because the user doesn't have to type the data; this approach avoids frustration because the user doesn't type the same value repeatedly, and it reduces document errors.

Notice that you change the value of a cell by modifying its `Formula` property. In this respect, a Visio shape acts precisely like a spreadsheet. However, the entry of text might look confusing at first. What you really need to enter is `="MyEntry"` as the formula. However, if you type that formula, Visio displays an error. You must always surround a formula with double quotes, so now the formula becomes `"="MyEntry""`. Unfortunately, this formula also raises an error because now VBA doesn't know what do to with `MyEntry` or how to concatenate it with the equals sign and blank string. To make VBA see a double quote as an internal part of a string, you must use two double quotes together so that the formula becomes `"=""MyEntry"""`. Figure BC2-5 shows the result of this code.

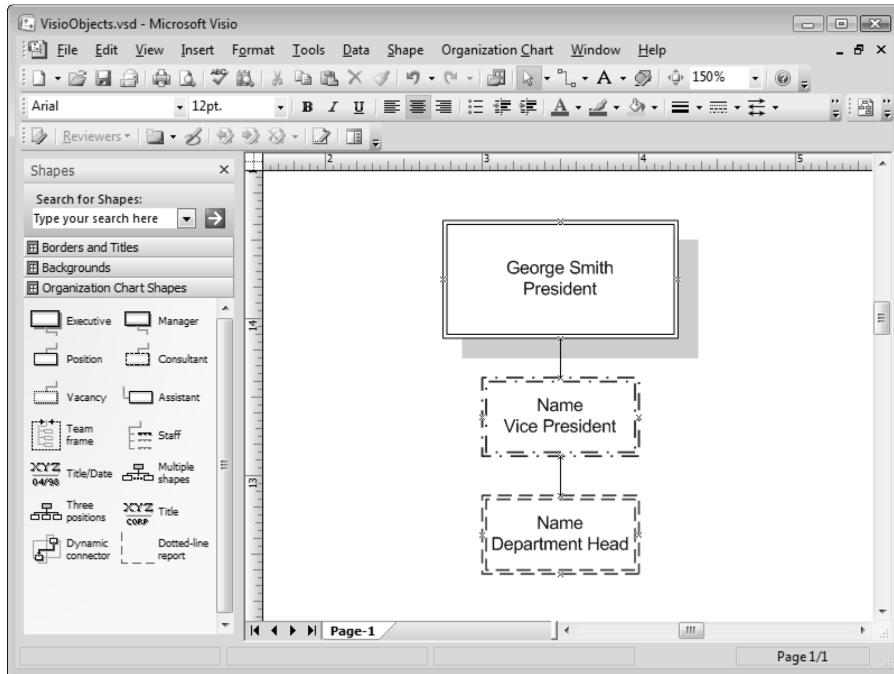


Figure BC2-5: You can manipulate shapes as needed through VBA code.

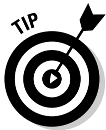
Sometimes a shape doesn't fall into a single category. In this case, the `Manager` shape can apply to vice presidents or department heads. You can still use this technique to avoid potential problems in consistency by displaying an appropriate dialog box to ask the user which kind of shape to create. The code uses a simple `MsgBox()` function, but you can use a custom form if you want.

The code sets the title, but not the name, for the shape. You still want the user to enter a name, and you could make this part of the form you create. The example relies on the user to perform the standard data entry in this case.

You can access any element of a shape's design by using VBA. The example changes the line color, pattern, and weight, as shown in Figure BC2-5, depending on the shape you choose. Notice how each of these entries requires a different formula to make the change. The `LineColor` index requires that you use the `=THEMEGUARD(RGB(128,0,0))` function. This function accepts a Red, Green, Blue (RGB) value as input, so you use the `RGB()` function to create this value.

The `LinePattern` formula requires a simple number as input. However, knowing which number to choose can be a problem. In this case, you can use the `MsoLineDashStyle` enumeration to obtain the value. Because the index names aren't documented, you often have to use the Object Browser to locate the information you need. This is where the search techniques described in Chapter 1 can come in very handy.

The `LineWeight` formula relies on what appears to be a string, but really isn't. In this case, you must provide a number and a unit of measure. The example uses the common unit of measure for this value, the *point,* which is 1⁄72 of an inch.



Of course, all these formula formats beg the question of how you figure them out if Microsoft hasn't documented any of this information anywhere. The approach I use is to write a simple three-line macro, as shown in Listing BC2-6.

Listing BC2-6 Obtaining a Formula

```
Sub TestFormula()
    ' Holds the test shape.
    Dim TestShape As Shape

    ' Choose the selected shape.
    Set TestShape = ActiveWindow.Selection(1)

    ' Display the desired cell.
    MsgBox TestShape.Cells("LineColor").Formula
End Sub
```

Now all I need to do is select a shape in the target page and run the macro to see the formula for the cell of interest. By varying the shape's properties, you can view the formula that will produce the effect you want. You can also check for this information by setting a breakpoint at the `MsgBox` line, typing **? TestShape.Cells("LineColor").Formula** in the Immediate window, and pressing Enter.