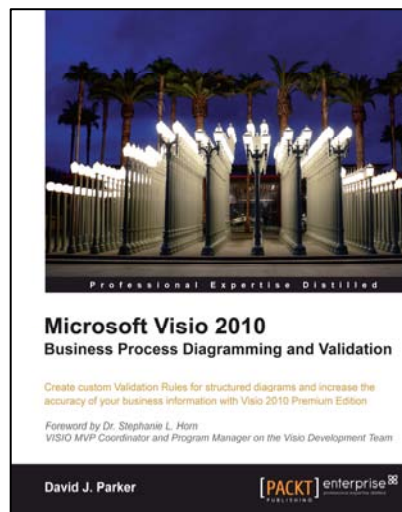


Microsoft Visio 2010 Business Process Diagramming and Validation

David J. Parker



Chapter No.2

"Understanding the Microsoft Visio Object Model"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "Understanding the Microsoft Visio Object Model"

A synopsis of the book's content

Information on where to buy this book

About the Author

David J. Parker explored linking Unix CAD and SQL databases in the early '90s for facilities and cable management, as he was frustrated as an architect in the late '80s, trying to match 3D building models with spreadsheets.

In '96 he discovered the ease of linking data to Visio diagrams of personnel and office layouts. He immediately became one of the first Visio business partners in Europe, and was soon invited to present his applications at worldwide Visio conferences. He started his own Visio-based consultancy and development business, **bVisual ltd** (<http://www.bvisual.net>), applying analysis, synthesis, and design to various graphical information solutions.

He has presented Visio solution provider courses for Microsoft EMEA, adding personal anecdotes and previous mistakes hoping that all can learn from them.

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

He wrote his first book, *Visualizing Information with Microsoft Office Visio 2007* (<http://www.visualizinginformation.com>), to spread the word about data-linked diagrams in business, and is currently writing his second book, which is about creating custom rules for validating structured diagrams in Visio 2010.

David wrote WBS Modeler for Microsoft, which integrates Visio and Project, and many other Visio solutions for various vertical markets.

David has been regularly awarded Most Valued Professional status for his Visio community work over the years, and maintains a Visio blog at <http://bvisual.spaces.live.com>. Based near to Microsoft UK in Reading, he still sees the need for Visio evangelism throughout the business and development community.

I would like to thank the Microsoft Visio for continuing to develop such a great application, and in particular, Stephanie Horn for agreeing to edit this book. Similarly, I would like to thank my fellow Visio MVP, John Marshall, for his help and encouragement. Most of all, I would like to thank my wife, Beena, for allowing me to write another book!

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

Microsoft Visio 2010 Business Process Diagramming and Validation

Once the creators of Aldus PageMaker had delivered Desktop publishing to the masses, they decided that they could make a smarter diagramming application. Eighteen months later, they emerged with the Visio product. Now they needed to get a foothold in the market, so they targeted the leading process flow diagramming package of the day, ABC FlowCharter, as the one to outdo. They soon achieved their aim to become the number one flowcharting application and so they went after other usage scenarios, such as network diagramming, organization charts, and building plans. In 1999, Microsoft bought Visio Corporation and Visio gradually became Microsoft Office Visio, meaning that all add-ons had to be written in a certain manner, and the common Microsoft Office core libraries like the Fluent UI were ever more increasingly employed.

Flowcharting still accounts for 30% of the typical uses that Visio is put to, but the core product did not substantially enhance its flowcharting abilities. There were some add-ons that provided rules, perhaps most notably for Data Flow Diagrams (which came and went); UML and Database Modelling, and many third parties have built whole flowcharting applications based on Visio. What all of these enhancements have in common is the imposition of a structure to the diagrams, which necessarily means the adoption of one rule set or another. There are a lot of competing and complementary rule sets in use, but what is important is that the chosen rule set fits the purpose it is being used for, and that it can be understood by other related professionals.

It is true that a picture is worth a thousand words, but the particular thousand words understood by each individual are more likely to be the same if the picture was created with commonly available rules. The structured diagramming features and Validation API in Visio 2010 enable business diagramming rules to be developed, reviewed, and deployed. The first diagramming types to have these rules applied to are process flowcharts, reminiscent of the vertical markets attacked by the first versions of Visio itself, but these rules can and will be extended beyond this discipline.

What This Book Covers

Chapter 1, Overview of Process Management in Microsoft Visio 2010, introduces the new features that have been added to Microsoft Visio to support structured diagrams and validation. You will see where Visio fits in the Process Management stack, and explore the relevant out of the box content.

Chapter 2, Understanding the Microsoft Visio Object Model, explains the Microsoft Visio 14.0 Type Library and the key objects, collections, and methods in the programmer's interface of Visio, where relevant for structured diagrams.

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

Chapter 3, Understanding the ShapeSheet™, explains the Microsoft Visio ShapeSheet™ and the key sections, rows, and cells, along with the functions available for writing ShapeSheet™ formulae, where relevant for structured diagrams.

Chapter 4, Understanding the Validation API, explains the Microsoft Visio Validation API and the key objects, collections, events, and methods in the programmer's interface for Visio diagram validation.

Chapter 5, Developing a Validation API Interface, is devoted to building a useful tool, called Rules Tools, to enable the tasks to be performed easily as Microsoft Visio 2010 does not provide a user interface to the Validation API for rules developers to use.

Chapter 6, Reviewing Validation Rules and Issues, will extend the tool, started in Chapter 5, to provide an import/export routine of rules to an XML file or to an HTML report, and a feature to add issues as annotations in Visio diagrams.

Chapter 7, Creating Validation Rules, will use the tool created in the previous chapter to create rules for structured diagramming. This chapter will look at common ShapeSheet™ functions that will be useful for rules, and the new Validation functions. It will also go through different scenarios for creating rules, especially with regard to Filter and Test Expressions.

Chapter 8, Publishing Validation Rules and Diagrams, will go through different methods for publishing Visio validation rules for others to use.

Chapter 9, A Worked Example for Data Flow Model Diagrams, presents a complete cycle for writing validation rules for the Data Flow Model Diagram methodology. Validation rules are created using the Rules Tools add-in developed in previous chapters, although alternative VBA code is provided.

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

2

Understanding the Microsoft Visio Object Model

Whatever programming language you code in, you need to understand the objects, properties, methods, relationships, and events of the application that you are working with. Without this knowledge, the development process is slow and any code you use is going to be inefficient. Visio is no different, in that it provides the **Visio Type Library** with all of its elements, but Visio also has a programmable ShapeSheet behind every shape. Therefore, the Visio Type Library can only be used efficiently if you understand the ShapeSheet, and in turn, the ShapeSheet formulae can only be used fully if you understand the Visio Type Library.

Also, if you are going to create validation rules to check the relationships and properties of structured diagrams, then you will need to understand how to traverse the Visio object model.

Therefore, this chapter is going to explain the **Microsoft Visio 14.0 Type Library** (`VisLib.dll`), and the key objects, collections, and methods in the programmer's interface of Visio, and the next chapter will reveal the ShapeSheet.

The Visio Type libraries

The publicly displayed version number of an application like Visio can be quite different from the internal version number that is revealed to programmers. For example, Microsoft Visio 2010 is the public version number for the internal version number 14. Therefore, programmers need to know that the Visio Type Library version is 14, although their users will know it as Visio 2010.

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book



There were no 13 versions prior to 14 because Visio was at version 6 (externally Visio 2000) when Microsoft bought the company in 1999. At that time, Microsoft Office was internally at version 9, so Microsoft Visio 2002 was internally hiked up to version 10 to be at the same version number as Microsoft Office 2002. At this point, Microsoft Visio 2003 was internally version number 11, and Microsoft Visio 2007 was internally 12. Version 13 went the same way as the thirteenth floors in high-rise buildings in the States—pandering to the superstitions of the masses.

Microsoft Visio 2010 may also install the following type libraries, depending upon the edition installed.

Name	File	Visio Editions
Microsoft Visio 14.0 Drawing Control Library	VisOcx.dll	All editions
Microsoft Visio 14.0 Save As Web Type Library	SaveAsWeb.dll	All editions
Microsoft Visio Database Modeling Engine Type Library	ModelEng.dll	Professional and Premium editions only
Microsoft Visio UML Add-In for Microsoft Visual C++ 6.0	UmlVC60.dll	Professional and Premium editions only
Microsoft Visio UML Solution for Visual Basic Type Library	UmlVB.dll	Professional and Premium editions only

In addition, since version 2007, Microsoft Outlook installs the Microsoft Visio Viewer (*vviewer.dll*), which has a useful programming interface itself. It allows pages, shapes, and data to be explored, even without Visio being installed. It is also available as a separate, free download from Microsoft, should you wish to use it on Windows desktops that do not have Microsoft Outlook installed.

But all I need is the object model

Some programmers think that Visio is present just to provide a graphical canvas with symbols and lines that they need to manipulate or interrogate. Perhaps they have been used to draw items in Windows Forms applications or even XAML-based development with WPF or Silverlight. To think like this is to misunderstand Visio because Visio has a rich diagramming engine, coupled with the ability to encapsulate data and custom behaviors in every element, not to mention the inheritance between certain types of objects. This has resulted in a fairly complex structure in parts of the object model, so that all of the desired functionality can be described fully.

Programmers who look at the Visio object model for the first time may be full of preconceptions and look in vain for the *x* and *y* coordinate of a shape on a page. They are surprised and a little frustrated that the *x* coordinate of a shape on a page is:

```
shape.CellsSRC(VisSectionIndices.visSectionObject,  
visRowIndices.visRowXFormOut,  
visCellIndices.visXFormPinX).ResultIU
```

The SRC part of the CellsSRC method is an acronym for **Section Row Column**, which will be explained later.

There is an alternative shorter form namely:

```
Shape.Cells("PinX").ResultIU
```

However, the shorter form is intrinsically more inefficient since the name has to be interpreted into the SRC indices by Visio anyway. Therefore, it is recommended that you work with the indices rather than the names, if at all possible.

The Visio object model is quite large, so I shall be selective by only discussing the parts that I think will assist in understanding and developing validation rules. There are other type libraries installed with Visio, but these are not relevant to the scope of this book. In addition, the Visio edition installed has an impact on the Visio type library itself. For example, the **Validation** objects and collections are only available if you have the **Premium** edition installed, and the **Data Linking** features are not available if you have only the **Standard** edition installed.

The other difference between the different Visio editions is the add-ons, templates, and stencils installed with it. But as these could be moved around and copied between users (illegally), their presence (or lack of presence) cannot be relied on to ascertain the edition installed. One way to ascertain the version is to check a specific registry setting (which is the only way if you are writing an installation script), or using the CurrentEdition property of the Application object.

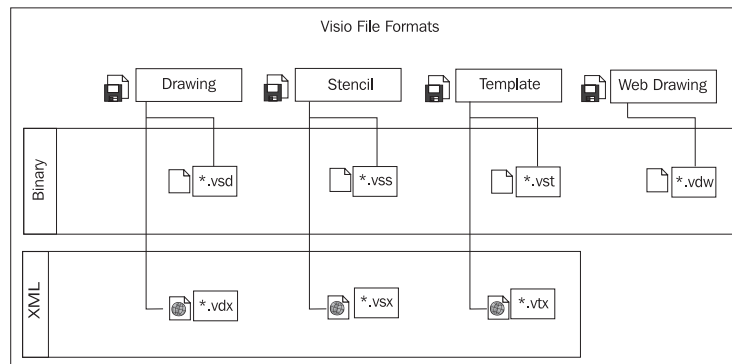
```
HKEY_CURRENT_USER\Software\Microsoft\Office\14.0\Visio\  
Application\LicenseCache
```

The expected values are STD, PRO, or PRM.

Types of Visio document

Before we get into the object model, we need to remind ourselves of the formats and types of Visio documents. Traditionally, Visio used its own binary format (which usually has an extension `*.vsd` for drawings), and then the XML format was introduced (`*.vdx` for drawings). The latter is approximately ten times larger in size than the former, although it often compresses to be smaller than the binary equivalent. The XML format is very verbose because it needs to describe the complexity of the graphics and the inheritance of elements within the document. In addition, it is not in the same zipped-up XML files in sub-folders format as most of the Microsoft Office applications.

The **Visio Web Drawing** is new in Visio 2010, which, when published to SharePoint 2010, allows certain elements that are linked to data recordsets to be automatically refreshed when the underlying data is updated, without using Visio. This Visio Services feature however, does not enable new shapes to be created, moved, or deleted, or for connections to be varied during the refresh. But it can be edited by the Visio client application to make these sorts of changes. This new file format has a `*.vdw` extension, and it contains **XAML** for rendering in Silverlight (or PNG format if required), in addition to the Visio document. These Visio files can be rendered by a new standard web part in Microsoft SharePoint 2010, which can be set to refresh either on a timer event, or manually.

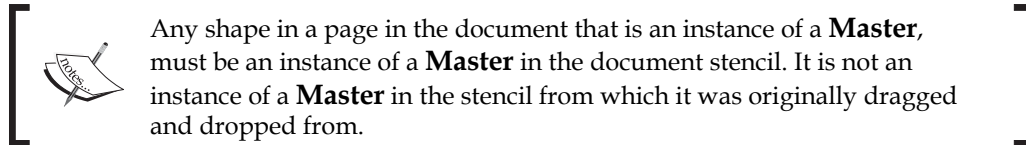


A Visio drawing document can save its workspace along with it, which usually means that there is a collection of docked stencils, which contain the shapes (properly referred to as **Masters** when they are in a stencil).

A Visio stencil is just a Visio document with the pages hidden, and is normally saved with a `*.vss` extension in the binary format, or `*.vsx` for the XML format.

A Visio template is just a Visio drawing document saved with a different extension, `*.vst` for binary and `*.vtx` for XML, so that Visio knows that the default action is to open a copy of it, rather than the original document.

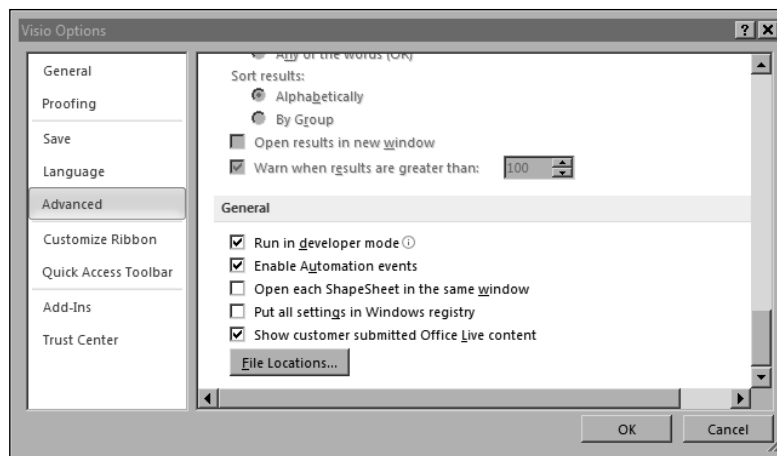
I mentioned that a stencil is just a Visio document with the drawing pages hidden. Well a drawing is just a Visio document which normally has its stencil hidden. However, you can reveal this in the UI with **More Shapes/Show Document Stencil**.



Which programming language should you use with Visio?

Microsoft Visio comes with **Visual Basic for Applications (VBA)** built into it, which is a very useful interface for exploring the object model, and testing out ideas. In addition, Visio has a macro recorder that can provide a quick and dirty way of exploring how some of the actions are performed. However, the resultant code from the macro recorder can be very verbose in parts, and completely miss out some bits because Visio is running code inside one of the many **Add-ons** or **COM add-ins** that may be installed.

If you want to use VBA then you will need to run Visio in **Developer Mode** by checking the option available from the **Visio Options** dialog (use **File | Options** to display this), in the **Advanced** group.



Developer Mode will also add some features to other parts of the Visio interface, such as additional options on the right-mouse menu when a page and shape is selected.



I have also formatted the output in the following code examples as a table for legibility, because the text will mostly wrap within the **Immediate Window**.

The Application object

The Application object is the root of most collections and objects in Visio, including the Active objects, two of which are useful for traversing structured diagrams – ActiveDocument and ActivePage.

Application
ActiveDocument
ActivePage
Addons
COMAddIns
CurrentEdition
DataFeaturesEnabled
Documents
TypelibMinorVersion
Version

The following sub-function in VBA prints out the salient information to the **Immediate Window**:

```
Public Sub DebugPrintApplication()
Debug.Print "DebugPrintApplication"
  With Visio.Application
    Debug.Print , "ActiveDocument.Name", .ActiveDocument.Name
    Debug.Print , "ActivePage.Name", .ActivePage.Name
    Debug.Print , "Addons.Count", .Addons.Count
    Debug.Print , "COMAddIns.Count", .COMAddIns.Count
    Debug.Print , "CurrentEdition", .CurrentEdition
    Debug.Print , "DataFeaturesEnabled", .DataFeaturesEnabled
    Debug.Print , "Documents.Count", .Documents.Count
    Debug.Print , "TypelibMinorVersion", .TypelibMinorVersion
    Debug.Print , "Version", .Version
  End With
End Sub
```

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

An example output is:

DebugPrintApplication	
ActiveDocument.Name	Visio Object Model.vsd
ActivePage.Name	The Application Object
Addons.Count	96
COMAddIns.Count	2
CurrentEdition	2
DataFeaturesEnabled	True
Documents.Count	7
TypelibMinorVersion	14
Version	14.0

The ActiveDocument and ActivePage objects

These objects can be referenced from the global object in VBA, but they are only available via the Application object in other languages.

The Addons collection

Microsoft writes all of its additional code as C++ add-ons to Visio as **Visio Solution Library files** (*.vs1), which are standard DLLs with specific header information in them. Others may write them as executable files (*.exe), which are generally slower because they are not running within the Visio process thread.

You can list the Add-ons that are loaded in your Visio installation like this:

```
Public Sub EnumerateAddons()  
Dim adn As Visio.Addon  
    Debug.Print "EnumerateAddons : Count = " &  
        Application.Addons.Count  
    Debug.Print , "Index", "Enabled", "NameU", "Name"  
    For Each adn In Application.Addons  
        With adn  
            Debug.Print , .Index, .Enabled, .NameU, .Name  
        End With  
    Next  
End Sub
```

This will output a very long list to your **Immediate Window**, the first few items are as follows:

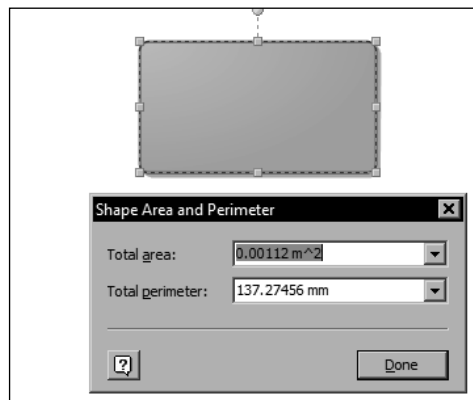
```
EnumerateAddons : Count = 96
```

Index	Enabled	NameU	Name
1	-1	Aec	Aec
2	-1	AutoSpaceConvert	AutoSpaceConvert
3	-1	AutoSpaceDrop	AutoSpaceDrop
4	-1	AutoSpaceResize	AutoSpaceResize
5	-1	Move Shapes...	Move Shapes...
6	-1	Shape Area and Perimeter...	Shape Area and Perimeter...
7	-1	Array Shapes...	Array Shapes...
8	-1	Measure Tool	Measure Tool
9	-1	BRAINSTORM	Brainstorming
10	-1	DB Engineer	DB Engineer
11	-1	DBWiz	Database Wizard

Note that the **NameU (Universal Name)** can be different than the **Name** property, although either can be used if you want to reference a particular add-on to run it. For example, if you select a shape in Visio, then type the following into the **Immediate Window**:

```
Application.Addons("Shape Area and Perimeter...").Run("")
```

This will cause the add-on to run, if you have a shape selected.



The COMAddIns collection

The `COMAddIns` collection is actually part of the Microsoft Office 14.0 Object Library, so you will need to set it correctly if you want **IntelliSense** to work in Visual Studio, or the VB Editor.

The following code will enumerate the loaded COMAddIns in your Visio application:

```
Public Sub EnumerateCOMAddIns()  
Dim adns As Office.COMAddIns  
Dim adn As Office.COMAddIn  
    Set adns = Application.COMAddIns  
    Debug.Print "EnumerateCOMAddIns"  
    Debug.Print , "Description"  
    For Each adn In adns  
        With adn  
            Debug.Print , .Description  
        End With  
    Next  
End Sub
```

The output in the **Immediate Window** will be something like this:

```
EnumerateCOMAddIns : Count = 2  
-----  
Description  
ValidationExplorer  
VisioAddIn1  
-----
```

The CurrentEdition property

Since the **Validation** object is only in Visio Premium edition, a further check could be included to ensure that `CurrentEdition` value is not `Standard` or `Professional`. It can be done using the following command:

```
If Application.CurrentEdition=visEdition.visEditionPremium Then  
....
```

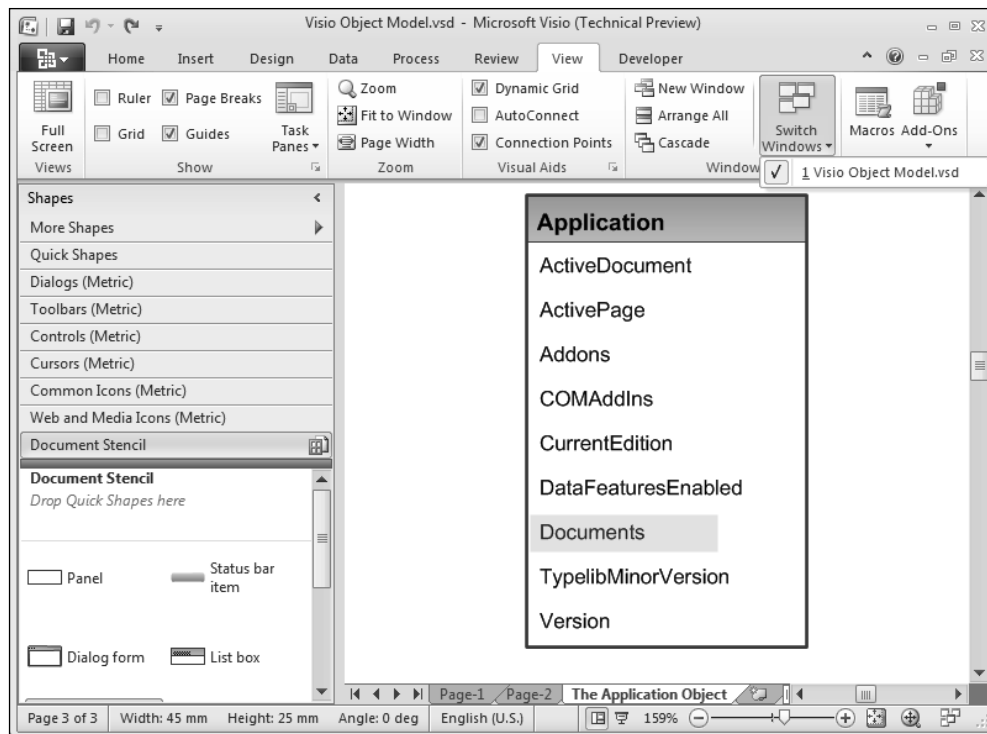
The DataFeaturesEnabled property

Data Linking and **Data Graphic** features are not available in Visio Standard, and they could be disabled in code, so you could check that this value is `True` if you want to interact with these particular features.

The Documents collection

The `Documents` collection contains all of the stencils and drawings that are currently open in the Visio application.

Consider this screenshot of a drawing that has been created from the **Software and Databases | Wireframe Diagram** template:



How many documents are open? Well, there is one showing, **Visio Object Model.vsd**, in the **Switch Windows** menu on the **View** tab. There appear to be seven docked stencils open too.

If you were to run the following code:

```
Public Sub EnumerateDocuments()
    Dim doc As Visio.Document
    Debug.Print "EnumerateDocuments : Count = " &
        Application.Documents.Count
    Debug.Print , "Index", "Type", "ReadOnly", "Name", "Title"
    For Each doc In Application.Documents
        With doc
            Debug.Print , .Index, .Type, .ReadOnly, .Name, .Title
        End With
    Next
End Sub
```


Then you might get output that looks like this:

EnumerateDocuments : Count = 7					
Index	Type	ReadOnly	Name	Title	
1	1	0	Visio Object Model.vsd	The Visio Object Model	
2	2	-1	WFDLGS_M.VSS	Forms and Dialogs	
3	2	-1	WFTLBR_M.VSS	Toolbars and Menus	
4	2	-1	WFCTRL_M.VSS	Controls	
5	2	-1	WFCRS_M.VSS	Cursors	
6	2	-1	WFCICN_M.VSS	Common Icons	
7	2	-1	WFWICN_M.VSS	Web and Media Icons	

As you can see, there are seven documents in all, one of which is `Type = 1` (**Drawing**) and the rest are `Type = 2` (**Stencil**). The **Document Stencil** is part of the Drawing, **Visio Object Model.vsd**.

The TypelibMinorVersion and Version properties

It may also be helpful to check the version of Visio, since **Validation** was not available prior to Visio 2010:

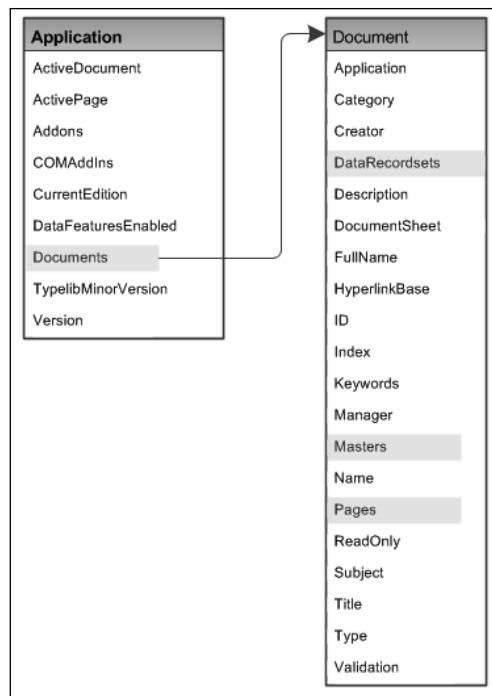
```
Application.Version = "14.0"
```

Or

```
Application.TypelibMinorVersion = 14
```

The Document object

The `Application.Documents` collection, seen highlighted in the following diagram contains many Document objects. The Document object contains the collections of `DataRecordsets`, `Masters`, `Pages`, and other properties, that you may need if you are validating a document.



The Advanced Properties object

The **Advanced Properties**, which are the document properties in the UI, could be referenced by the **Validation** expressions, as follows:

- Category
- Creator displayed as **Author** in the **Properties** dialog
- Description displayed as **Comments** in the **Properties** dialog
- HyperlinkBase
- Keywords displayed as **Tags** in the **Properties** dialog
- Manager
- Subject
- Title

You can view these values in the backstage panel, and in the **Advanced Properties** option on the **Properties** button.

```
Public Sub DebugPrintDocumentAdvancedProperties()  
    Debug.Print "DebugPrintDocumentAdvancedProperties : " &  
        ActiveDocument.Name  
    With ActiveDocument  
        Debug.Print , "Title", .Title  
        Debug.Print , "Subject", .Subject  
        Debug.Print , "Author", .Creator  
        Debug.Print , "Manager", .Manager  
        Debug.Print , "Company", .Company  
        Debug.Print , "Language", .Language  
        Debug.Print , "Categories", .category  
        Debug.Print , "Tags", .Keywords  
        Debug.Print , "Comments", .Description  
        Debug.Print , "HyperlinkBase", .HyperlinkBase  
    End With  
End Sub
```

The output would be as follows.

```
DebugPrintDocumentAdvancedProperties : Partial Visio Object Model  
and VBA Code.vsd
```

Title	The Visio Object Model
Subject	Business Process Diagramming in Visio 2010
Author	David J Parker
Manager	Stephanie Moss
Company	bVisual ltd
Language	1033
Categories	Samples
Tags	Visio, Object Model, Type Library
Comments	This document contains sample VBA code
HyperlinkBase	http://www.bvisual.net

The DataRecordsets collection

If you are using the **Data Linking** features, then you may want to reference one or more of the DataRecordsets objects in the document.

```
Public Sub EnumerateRecordsets()  
    Dim doc As Visio.Document  
    Dim dst As Visio.DataRecordset  
    Set doc = Application.ActiveDocument  
    Debug.Print "EnumerateRecordsets : Count = " &  
        doc.DataRecordsets.Count  
    Debug.Print , "ID", "DataConnection", "Name"
```

```

    For Each dst In doc.DataRecordsets
        With dst
            Debug.Print , .ID, .DataConnection, .Name
        End With
    Next
End Sub

```

The output from the above will be similar to this:

EnumerateRecordsets : Count = 1		
ID	DataConnection	Name
2	2	XLEXTDAT9 DemoData NetworkStatus

Note that the **Pivot Diagram** feature in Visio creates multiple `DataRecordsets` which are not visible in the normal UI.

The DocumentSheet object

The `DocumentSheet` object is the **ShapeSheet** of `Documents`.

If you wanted to ensure that a document is uniquely identifiable, since its name can be changed, then you can use the `UniqueID` property to generate a **GUID** for the `DocumentSheet`, for example where `doc` is a `Document` object.

```
doc.DocumentSheet.UniqueID(VisUniqueIDArgs.visGetOrMakeGUID)
```

The ID and Index properties

An `ID` is assigned to a document when it is added to the `Documents` collection, and it will be kept so long as the document stays open, whereas the `Index` may change if other documents are closed.

The FullName and Name properties

The `Name` property is the file name without the path, whilst the `FullName` is the whole path, including the `Name`.

The Masters collection

The Document object contains the Masters collection.

```
Public Sub EnumerateMasters()  
Dim doc As Visio.Document  
Dim mst As Visio.Master  
Set doc = Application.ActiveDocument  
Debug.Print "EnumerateMasters : Count = " & doc.Masters.Count  
Debug.Print , "ID", "Type", "OneD", "Hidden", "Name"  
For Each mst In doc.Masters  
With mst  
Debug.Print , .ID, .Type, .OneD, .Hidden, .Name  
End With  
Next  
End Sub
```

This code will produce output similar to the following:

EnumerateMasters : Count = 3				
ID	Type	OneD	Hidden	Name
6	1	0	0	List box
7	1	0	0	List box item
9	1	-1	0	Dynamic connector

The Type=1 is the constant `visMasterTypes.visTypeMaster`. There are other types for fills, themes, and data graphics but they will usually be hidden to ensure that the user does not accidentally drag-and-drop them off the document stencil in the UI.

The Pages collection

The Pages collection of the Document object contains all pages in the document, regardless of type, thus you may need to filter by type when you are traversing them.

The following code provides a simple enumeration of the pages:

```
Public Sub EnumeratePages()  
Dim doc As Visio.Document  
Dim pag As Visio.Page  
Set doc = Application.ActiveDocument  
Debug.Print "EnumeratePages : Count = " & doc.Pages.Count  
Debug.Print , "Index", "ID", "Type", "Name"  
For Each pag In doc.Pages  
With pag  
Debug.Print , .Index, .ID, .Type, .Name  
End With  
Next  
End Sub
```

```

        End With
    Next
End Sub

```

The output will be similar to this:

EnumeratePages : Count = 4			
Index	ID	Type	Name
1	5	1	The Application Object
2	0	1	Page-1
3	4	1	Page-2
4	6	1	Page-4

Notice that the ID property does not need to be contiguous!

The ReadOnly property

This is a Boolean (True/False) property.

The Type property

You can test for the type of document in code to ensure that it is the type that you want.

```

If doc.Type=VisDocumentTypes.visTypeDrawing Then
...

```

The other types are `visTypeStencil` and `visTypeTemplate`.



The Validation object

The **Validation** object provides access to the **Validation API** and will be discussed at length in *Chapter 4*.

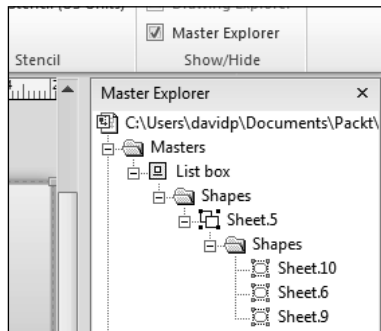
The Master object

When a **Master** shape is dragged-and-dropped from a stencil onto a page, (or by using any of the `PageDrop` methods) then Visio checks the local document stencil to see if the master already exists.

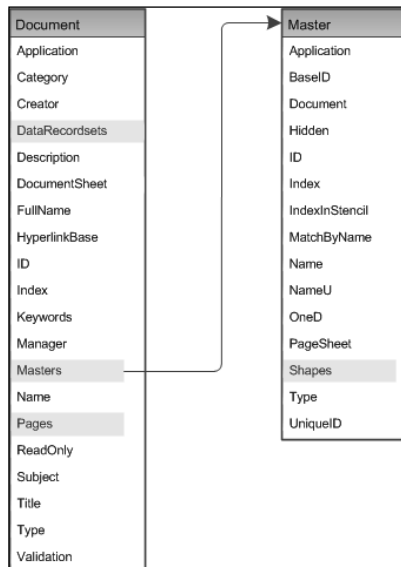
If a master name exists already and it has not been edited locally, or, even if it has and the `MatchByName` property is `true`, then the shape becomes an instance of the local master. If it does not exist, then the master is copied from the docked stencil to the local stencil, so that the shape can become an instance of it.

 The `MatchByName` property can be set by editing a master's properties in the user interface, and changing the **Match Master By Name on Drop** checkbox in the **Master Properties** dialog. 

If you open a **Master** on your local document stencil via **Edit Master | Edit Master Shape**, then you can open the **Master Explorer** window. You can then see that it is usually composed of a single **Shape** which often has a **Shapes** collection within it.



You can do a certain amount of editing to the shape in a local master, and have these changes propagated to all instances within the document. However, many users make the assumption that you can simply replace the master in a document to update the instances. This is not so, although some third-parties have attempted to make tools that can perform this task.



For More Information:
www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

The BaseID property

It is possible that many `Masters` have been derived from the same root `Master`, in which case they would all have the same `BaseID`.

The Hidden property

If this value is `true`, then the `Master` is hidden in the UI, but it still can have shape instances. This is merely the display position of the `Master` in the stencil.

The ID, Index, and IndexInStencil properties

An `ID` is assigned to a master when it is added to the `Masters` collection, and it will be kept so long as the document exists. The `Index` is the read-only ordinal position in the stencil, but the `IndexInStencil` controls the display position in the stencil, and can be modified.

The Name and NameU properties

The `Name` property is the displayed name, which could be different to the universal `NameU` property.

The PageSheet object

The `PageSheet` object is the `ShapeSheet` of the `Master` (or a `Page`).

If you wanted to ensure that a page is uniquely identifiable, since its name can be changed, then you can use the `UniqueID` property to generate a GUID for the `PageSheet`, for example, where `pag` is a `Page` object.

```
pag.PageSheet.UniqueID(VisUniqueIDArgs.visGetOrMakeGUID)
```

The Type property

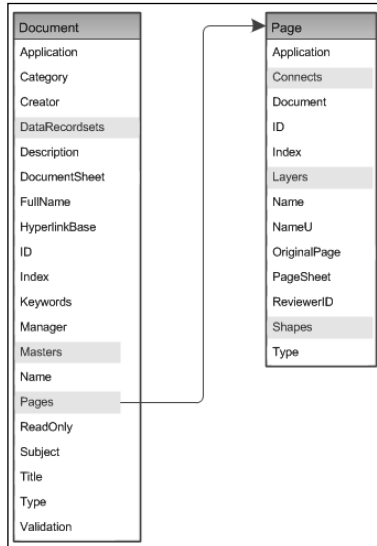
There are many different types of `Master`, since they are used to define **Data Graphics, Fills, Lines, and Themes** so it can be useful to check first.

```
If master.Type = Visio, visMasterTypes.visTypeMaster Then
...

```


The Page object

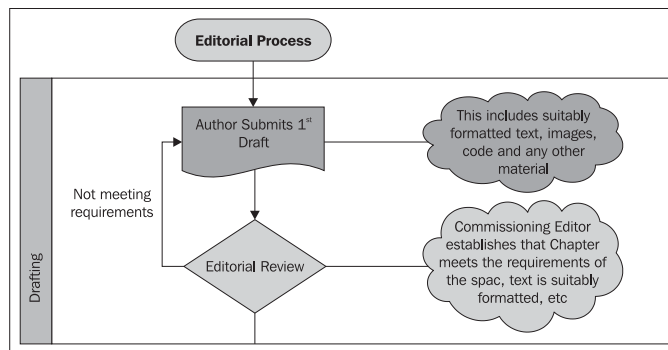
The Page object contains the `Connects`, `Layers`, and most importantly, the `Shapes` collections.



The Connects collection

The page has a `Connects` collection that contains all of the shape connections in it. A developer can now use the simpler `ConnectedShapes` and `GluedShapes` methods, described later in this chapter, but it is worth understanding this collection.

In a process diagram, most flowchart shapes are connected to each other via a **Dynamic Connector** shape. So, each **Dynamic Connector** (which is **OneD**) shape is usually connected to a flowchart shape at each end of it. The cell at the start of the line is called **BeginX**, and the cell at the end is called **EndX**.



You can iterate the `Connects` collection with the following code:

```
Public Sub EnumeratePageConnects()
Dim pag As Visio.Page
Dim con As Visio.Connect
Set pag = Application.ActivePage
Debug.Print "EnumeratePageConnects : Count = " &
pag.Connects.Count
Debug.Print , "Index", "FromSheet.Name", "FromCell.Name",
"FromSheet.Text ", _
"ToSheet.Name", "ToCell.Name", "ToSheet.Text"
For Each con In pag.Connects
With con
Debug.Print , .Index, .FromSheet.Name, .FromCell.Name,
.FromSheet.Text, _
.ToSheet.Name, .ToCell.Name, .ToSheet.Text
End With
Next
End Sub
```

This is the first few rows of the example output:

EnumeratePageConnects : Count = 32						
Index	FromSheet. Name	FromCell. Name	FromSheet. Text	ToSheet. Name	ToCell. Name	ToSheet. Text
1	Dynamic connector	BeginX		Start/End	PinY	Editorial Process
2	Dynamic connector	EndX		Document	PinY	Author Submits 1st Draft
3	Dynamic connector.5	BeginX		Document	PinY	Author Submits 1st Draft
4	Dynamic connector.5	EndX		Decision	PinY	Editorial Review
5	Dynamic connector.7	BeginX	Pass	Decision	PinX	Editorial Review
6	Dynamic connector.7	EndX	Pass	Process	PinX	1st Draft Peer Reviewed

I have displayed the text on each shape to make it easier to understand, but it is more likely that you will need to read the **Shape Data** on each shape in more complex diagrams.

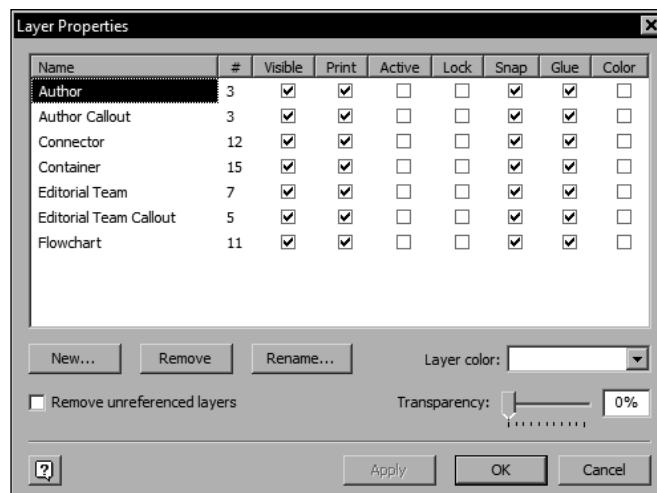
The ID and Index properties

An ID is assigned to a page when it is added to the Pages collection, and it will be kept, whereas the Index will change if the page order is modified.

The Layers collection

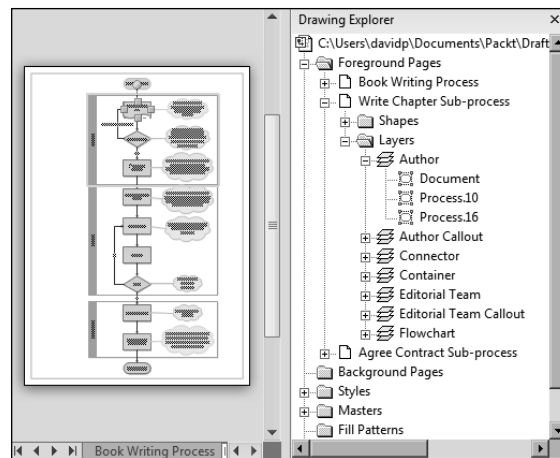
A page can contain many layers, which can have their Visible and Print setting toggled, amongst other options. However the display in Silverlight (as contained in the Visio drawing for web format) does not respect any of these settings. This is probably because a Visio shape can belong to none or many layers, making the correlation to XAML very difficult.

Users often confuse layers with the display order in the Z-order or index. The Z-index is controlled by the index of the shape within the page. The Move Forwards, Move To Front, Move Backwards, and Move to Back commands merely change the index of the affected shapes. However, Visio 2010 has introduced a new way to control the display level, which will be discussed in the next chapter.



The sum of the number of shapes on each layer can be less or greater than the total number of shapes on a page because a shape can belong to none or multiple layers, and shapes with subshapes can have different layer membership.

The **Drawing Explorer** window provides an easy way of viewing the list of shapes assigned to each layer.



You can iterate the layers on a page in code:

```
Public Sub EnumeratePageLayers()
    Dim pag As Visio.Page
    Dim lyr As Visio.Layer
    Set pag = Application.ActivePage
    Debug.Print "EnumeratePageLayers : Count = " & pag.Layers.Count
    Debug.Print , "Index", "Row", "Visible", "Print", "Name"
    For Each lyr In pag.Layers
        With lyr
            Debug.Print , .Index, .Row,
                .CellsC(VisCellIndices.visLayerVisible),
                .CellsC(VisCellIndices.visLayerPrint), .Name
        End With
    Next
End Sub
```

This could provide output like this:

EnumeratePageLayers : Count = 7				
Index	Row	Visible	Print	Name
1	0	1	1	Flowchart
2	1	1	1	Connector
3	2	1	1	Author
4	3	1	1	Editorial Team
5	4	1	1	Author Callout
6	5	1	1	Editorial Team Callout
7	6	1	1	Container

Layers are useful for controlling visibility of shapes assigned to them, and they provide a way of retrieving a selection of shapes. They can also be part of a validation expression.

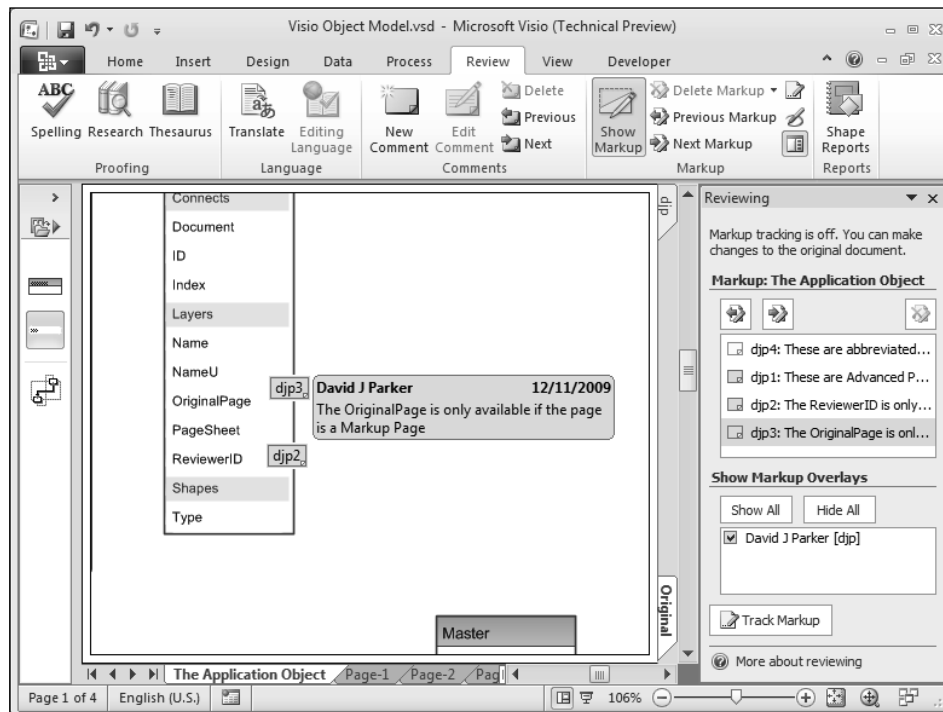
The PageSheet object

The PageSheet object is the ShapeSheet of the Master (or a Page. See *The Master object* section covered previously).

The Reviewer property

When a user tracks markup using the **Review** tab in Visio on a page, then a new page is added to the document Pages collection. This new page provides a canvas for adding comments and any shapes over the top of the original page, but without adding anything to the original page.

If a page is a markup page, then the ReviewerID property is available in code. The ReviewerID is an index into specific cells in the DocumentSheet, as you will discover in the next chapter.



The following code can be run on a normal (non-markup) page:

```

Public Sub EnumeratePageMarkups()
Dim pag As Visio.Page
Dim pagTest As Visio.Page
Set pag = Application.ActivePage
Debug.Print "UserName : " & pag.Application.Settings.UserName
Debug.Print "UserInitials : " &
    pag.Application.Settings.UserInitials
Debug.Print "EnumeratePageMarkups for " & pag.Name

Debug.Print , "Index", "ID", "ReviewerID", "Name"
For Each pagTest In pag.Document.Pages
    With pagTest
        If .Type = VisPageTypes.visTypeMarkup Then
            If .OriginalPage Is pag Then
                Debug.Print , .Index, .ID, .ReviewerID, .Name
            End If
        End If
    End With
Next
End Sub

```

This will produce the following output:

```

UserName : David J Parker
UserInitials : djp
EnumeratePageMarkups for The Application Object

```

Index	ID	ReviewerID	Name
5	7	1	The Application Object [djp]

Notice that the markup page name is the same as the `OriginalPage.Name`, but with the reviewers initials appended in square brackets.

It is possible to iterate through the comments too, but this requires some understanding of the `ShapeSheet`, which comes in the next chapter.



Comments are not displayed in the new Microsoft SharePoint Web Part, which displays the Visio document for the web format (*.vdw).

The Shapes collection

Each Page, Master, or Shape can have a Shapes collection. The Shapes collections contains all of the shapes, whether they are instances of a Master, or simple drawn lines, rectangles, text, and so on.

In this example, I have simply shown how to iterate through the shapes on a page.

```
Public Sub EnumeratePageShapes()  
Dim pag As Visio.Page  
Dim shp As Visio.Shape  
Set pag = Application.ActivePage  
Debug.Print "EnumeratePageShapes : Count = " & pag.Shapes.Count  
Debug.Print , "Index", "ID", "Type", "OneD", "Is Instance",  
"Name", "Text"  
For Each shp In pag.Shapes  
With shp  
Debug.Print , .Index, .ID, .Type, .OneD, Not .Master Is  
Nothing, .Name, .Text  
End With  
Next  
End Sub
```

Here are a few lines from the output as follows:

EnumeratePageShapes : Count = 35						
Index	ID	Type	OneD	Is Instance	Name	Text
1	34	2	0	True	Container 3	Drafting
2	39	2	0	True	Container 3.39	Editing
3	44	2	0	True	Container 3.44	Production
4	20	5	0	False	Sheet.20	
5	1	3	0	True	Start/End	Editorial Process
6	2	3	0	True	Document	Author Submits 1st Draft
7	3	3	-1	True	Dynamic connector	
8	4	3	0	True	Decision	Editorial Review

It may be necessary to test that specific shapes exist on a page during the validation process. For example, it may be a requirement that there is a Start and End flowchart shape.

The Type property

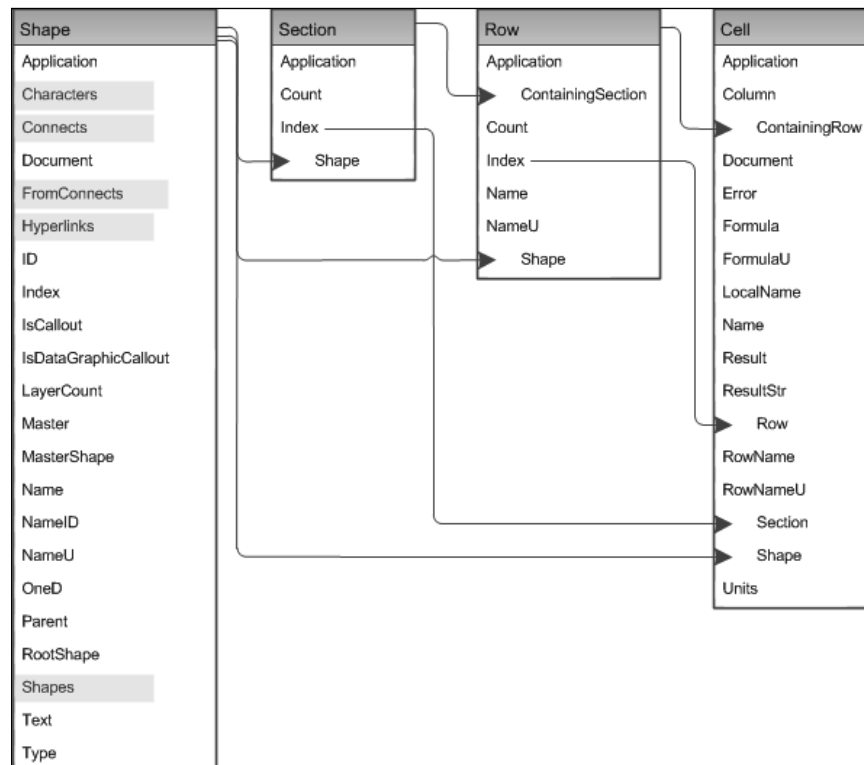
There are several types of Page in Visio, namely **Foreground**, **Background** and **Markup**. Any page in Visio can have an associated Background page, and any number of associated Markup pages used by reviewers. Therefore, it is usual to check the page type in code before continuing with any operations on it.

```
If pag.Type = visPageTypes.visTypeForeground Then
...

```

The Shape object

The Shape object is the most important object in the Visio application, and it needs to be seen as a whole with its member Sections, Rows, and Cells to understand its complexity.



Here is a function that prints out basic information about a selected shape into the **Immediate Window in VBA**:

```
Public Sub DebugPrintShape()  
If Application.ActiveWindow.Selection.Count = 0 Then  
    Exit Sub  
End If  
Dim shp As Visio.Shape  
Set shp = Application.ActiveWindow.Selection.PrimaryItem  
Debug.Print "DebugPrintShape : " & shp.Name  
With shp  
    Debug.Print , "Characters.CharCount", .Characters.CharCount  
    Debug.Print , "Connects.Count", .Connects.Count  
    Debug.Print , "FromConnects.Count", .FromConnects.Count  
    Debug.Print , "Hyperlinks.Count", .Hyperlinks.Count  
    Debug.Print , "ID", .ID  
    Debug.Print , "Index", .Index  
    Debug.Print , "IsCallout", .IsCallout  
    Debug.Print , "IsDataGraphicCallout", .IsDataGraphicCallout  
    Debug.Print , "LayerCount", .LayerCount  
    Debug.Print , "Has Master", Not .Master Is Nothing  
    Debug.Print , "Has MasterShape", Not .MasterShape Is Nothing  
    Debug.Print , "Name", .Name  
    Debug.Print , "NameID", .NameID  
    Debug.Print , "NameU", .NameU  
    Debug.Print , "OneD", .OneD  
    Debug.Print , "Parent.Name", .Parent.Name  
    Debug.Print , "Has RootShape", Not .RootShape Is Nothing  
    Debug.Print , "Text", .Text  
    Debug.Print , "Type", .Type  
End With  
End Sub
```

This produces the following output in my sample workflow as follows, when the Document shape with the text Author Submits 1st Draft is selected before the code is run:

DebugPrintShape : Document	
Characters.CharCount	24
Connects.Count	0
FromConnects.Count	3
Hyperlinks.Count	0
ID	2

DebugPrintShape : Document	
Index	6
IsCallout	False
IsDataGraphicCallout	False
LayerCount	2
Has Master	True
Has MasterShape	True
Name	Document
NameID	Sheet.2
NameU	Document
OneD	0
Parent.Name	Write Chapter Sub-process
Has RootShape	True
Text	Author Submits 1st Draft
Type	3

The Characters and Text properties

Every shape in Visio has a text block, regardless of whether there are any characters in it. This text block can be multiple lines, contain different fonts and formats, and can even contain references to other cell values. Indeed, if a text block does contain references to other cells, then the `shape.Text` property in code will display special characters instead of the actual value. However, `shape.Characters.Text` will return the referenced cell's values. Therefore, it is usually better to use the `shape.Characters.Text` property.

The Connects and FromConnects collections

The `Connects` collection contains the connections that the source shape is connected to, whereas the `FromConnects` collection contains the connections that are connected to the source shape.

Sounds easy, but it isn't. Traversing a structured diagram using these collections gets terribly messy, so use the newly added `ConnectedShapes` and `GluedShapes` methods, as described in the *Connectivity API* section covered later in this chapter.

The Hyperlinks collection

Hyperlinks can be created in the UI, in code, or even automatically by using **Data Linking**. Hyperlinks can contain `http:`, `https:`, and even `mailto:` URLs. Therefore, you may need to be aware of, and even report on them.

The ID, Index, NameID, Name, and NameU properties

The `Index` is controlled by the Z-index or Z-order in the user interface (by using **Send To Back**, **Bring to Front**, and so on), whereas the `ID` is a sequential number that is assigned when the shape is created. The `NameID` is concatenation of `Sheet` and `ID`.

The `Name` and `NameU` are automatically created, usually as a concatenation of the `Master.Name` and `ID`, and are originally identical. These properties can be modified (even independently of each other), but they must be unique for the `Shapes` collection of the parent. The `NameU` is the `Shapes`' locale-independent name, but `Name` can be locale-specific.

The `IsCallout` and `IsDataGraphicCallout` properties

The `IsCallout` property is a new property for Visio 2010, implemented so that you can spot more easily if a shape is one of the new callout shapes. The `IsDataGraphicCallout` property was introduced in Visio 2007 so that you can identify if the parent shape is a **Data Graphic** shape.

The `LayerCount` property

A shape can be a member of none, one, or multiple layers, which can lead to great complexity. You may wish to have a rule that a shape must only belong to a single layer.

The `Master`, `MasterShape`, and `RootShape` objects

A shape in Visio can either be an instance of a `Master`, that is one that has been dragged-and-dropped from a stencil, or it is one that is just drawn, like a line, rectangle, ellipse, or text. You can test this by checking if the `shape.Master` or `shape.MasterShape` object exists (`Is Nothing`) or not.

If the shape is part of a `Master` instance, then the `RootShape` is the top-level shape of the instance.

The `OneD` property

The `OneD` property is `true` if the shape is set to behave like a line.

The `Parent` object

The `Parent` property is never `Nothing`, but it can be either a `Page`, `Master`, or `Shape`.

Note that the `Parent` object may also be one of the following `Containing` properties:

- A shape in `Page.Shapes` collection always has values for the `ContainingPage` and `ContainingPageID` properties
- A shape in `Master.Shapes` collection always has values for the `ContainingMaster` and `ContainingMasterID` properties
- A shape in `Shape.Shapes` collection always has values for the `ContainingShape` and `ContainingShapeID` properties

The Type property

A shape can be a group of other shapes, in which case the `shape.Type` property will be equal to `VisShapeTypes.visTypeGroup` and the `shape.Shapes` collection will probably contain other shapes.

There are other shape types too, such as **Guide** and **Ink**, but most will be `VisShapeTypes.visTypeShape` or `VisShapeTypes.visTypeGroup`.

The Section object

Visio ShapeSheets have two types of Sections—fixed and variable. You can always rely upon a fixed `Section` being present, thus you do not need to test for its existence before referencing it.

However, some sections are optional (and in the case of **Geometry**, there may be multiple occurrences). Therefore, you may need to test for their existence before referencing them. The most common variable sections that you will need to be aware of are for **Shape Data**, **User-defined Cells**, and less often, **Hyperlinks**. You will learn more about these in the next chapter.

Use the enum `VisSectionIndices` in the **Visio Type Library** to get the right integer value for the `Section.Index` property. For example, you could test for the presence of a **Shape Data** section in a shape as follows (where `shp` is a **Shape** object):

```
If shp.SectionExists(VisSectionIndices.visSectionProp, VisExistsFlags.visExistsAnywhere) Then...
```

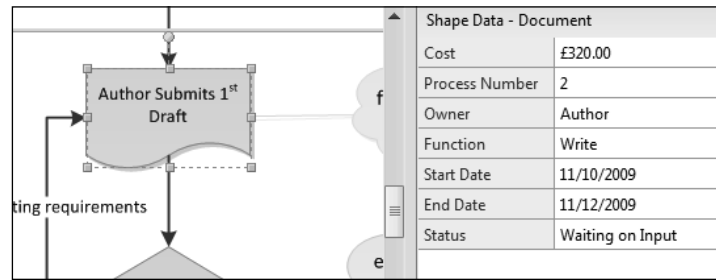
You can get the number of Rows in a Section using the `RowCount` method as follows:

```
For i = 0 to shp.RowCount(VisSectionIndices.visSectionProp) - 1...
```

The Row object

Sections contain **Rows**, just like a worksheet in Excel, and each Row contains cells. All of the interesting information is at the `Cell` object level.

Take this example where I have selected a Document shape.



I can enumerate through the cells of the **Shape Data** section using the following code:

```
Public Sub EnumerateShapePropRows ()
    If Application.ActiveWindow.Selection.Count = 0 Then
        Exit Sub
    End If
    Dim shp As Visio.Shape
    Dim iRow As Integer
    Dim cel As Visio.Cell
    Set shp = Application.ActiveWindow.Selection.PrimaryItem
    Debug.Print "EnumerateShapePropRows : " & shp.Name
    If Not shp.SectionExists (VisSectionIndices.visSectionProp,
        VisExistsFlags.visExistsAnywhere) Then
        Debug.Print , "Does not contain any Shape Data rows"
        Exit Sub
    End If
    With shp
        Debug.Print , "Shape Data row count : ",
            .RowCount (VisSectionIndices.visSectionProp)
        Debug.Print , "Row", "RowName", "Label"
        For iRow = 0 To .RowCount (VisSectionIndices.visSectionProp) - 1
            Set cel = .CellsSRC (VisSectionIndices.visSectionProp,
                iRow, 0)
            Debug.Print , cel.Row, cel.RowName,
                .CellsSRC (VisSectionIndices.visSectionProp, iRow,
                    VisCellIndices.visCustPropsLabel) .ResultStr ("")
        Next iRow
    End With
End Sub
```

This will produce the following output:

EnumerateShapePropRows : Document		
Shape Data row count :		7
Row	RowName	Label
0	Cost	Cost
1	ProcessNumber	Process Number
2	Owner	Owner
3	Function	Function
4	StartDate	Start Date
5	EndDate	End Date
6	Status	Status

Notice that I had to use the `CellsSRC()` method to iterate through the `Row`, and that I need to understand what values to use for the third parameter.

Moreover, I know that the `RowName` is safe to use on the **Shape Data** section, but some **Sections** do not have names for their **Rows**.

I have also displayed the difference between the `RowName` and the `Label` of a **Shape Data** row. Note that the `RowName` cannot contain any special characters or spaces, whereas `Label` can.

The Cell object

We must look a little more closely at the `Cell` object.

Cell
Application
Column
ContainingRow
Document
Error
Formula
FormulaU
LocalName
Name
Result
ResultStr
Row
RowName
RowNameU
Section
Shape
Units

The Column property

There are a different number of columns in different **Sections** of ShapeSheet. Therefore, you should use the `Section` specific values of the `VisCellIndices` enum to refer to a specific cell column. For example, the **User-defined Cells** section column indices begin with `visCellIndices.visUser`. However, all of the **Shape Data** section column indices begin with `visCellIndices.visCustProps` because **Shape Data** used to be called **Custom Properties**.

The Error property

If a `Cell` formula is unable to evaluate, then the `Error` value is one of the `VisCellError` enum values. This value is generated along with the result.

The Formula and FormulaU properties

Every `Cell` in Visio can contain a formula. This formula can contain references to other cells, and because Visio works with multiple languages the `Formula` string is the localized version of the `FormulaU` string, which is in English.

The Name and LocalName properties

For some languages, the `LocalName` property may be different to the English `Name` property.

The Result properties

There are quite a few different cell properties that begin `.Result` because the data type is agnostic. Generally, you can retrieve text values using the `.ResultStr("")` property, and numeric values using the `.ResultIU` property. **IU** stands for **Internal Units** in this case, but you could also use the `.Result("m")` property to return a numeric property formatted in the units of your choice.

Also, be aware that there is a powerful `Application.ConvertResult` method that you can use to convert values between units.

The Units property

This is an integer value from the `VisUnitCodes` enum.

Iterating through cells

Now that we understand a bit more about the `Cell` object, we can iterate through some cells in the **Shape Data** rows of a selected shape:

```
Public Sub EnumerateShapePropCells()  
If Application.ActiveWindow.Selection.Count = 0 Then  
    Exit Sub  
End If  
Dim shp As Visio.Shape  
Dim iRow As Integer  
Dim iCol As Integer  
Dim cel As Visio.Cell  
Set shp = Application.ActiveWindow.Selection.PrimaryItem  
Debug.Print "EnumerateShapePropRows : " & shp.Name  
If Not shp.SectionExists(VisSectionIndices.visSectionProp,  
    VisExistsFlags.visExistsAnywhere) Then  
    Debug.Print , "Does not contain any Shape Data rows"  
    Exit Sub  
End If  
With shp  
    Debug.Print , "Shape Data row count : ",  
        .RowCount(VisSectionIndices.visSectionProp)  
    Debug.Print , "Row", "RowName"  
    Debug.Print , , "Column", "Cell.Name", "Cell.Formula",  
        "Cell.ResultIU", "Cell.ResultStr( "" )"  
    For iRow = 0 To .RowCount(VisSectionIndices.visSectionProp) - 1  
        For iCol = 0 To  
            .RowsCellCount(VisSectionIndices.visSectionProp, iRow) - 1  
            Set cel = .CellsSRC(VisSectionIndices.visSectionProp,  
                iRow, iCol)  
            Debug.Print , , iCol, cel.Name, cel.Formula,  
                cel.ResultIU, cel.ResultStr( "" )  
        Next iCol  
    Next iRow  
End With  
End Sub
```


On my selected Document shape, the top of the output looks like this:

EnumerateShapePropRows : Document

Shape Data row count : 7

Row	ColumnName	Cell.Name	Cell.Formula	Cell.ResultIU	Cell.ResultStr("")
0		Prop.Cost	CY(320,"GBP")	320	£320.00
1		Prop.Cost.Prompt	" "	0	
2		Prop.Cost.Label	"Cost"	0	Cost
3		Prop.Cost.Format	"@"	0	@
4		Prop.Cost.SortKey	" "	0	
5		Prop.Cost.Type	7	7	7
6		Prop.Cost.Invisible	FALSE	0	FALSE
7		Prop.Cost.Verify	FALSE	0	FALSE
8		Prop.G7	0	FALSE	
9		Prop.H7	0	FALSE	
10		Prop.I7	0	FALSE	
11		Prop.J7	0	FALSE	
12		Prop.K7	0	FALSE	
13		Prop.L7	0	FALSE	
14		Prop.Cost.LangID	1033	1033	1033
15		Prop.Cost.Calendar	0	0	0

Cells 8 through 13 stick out because they do not appear in the UI at all. In fact, these are reserved for internal use or future use by Microsoft, so use them at your peril!

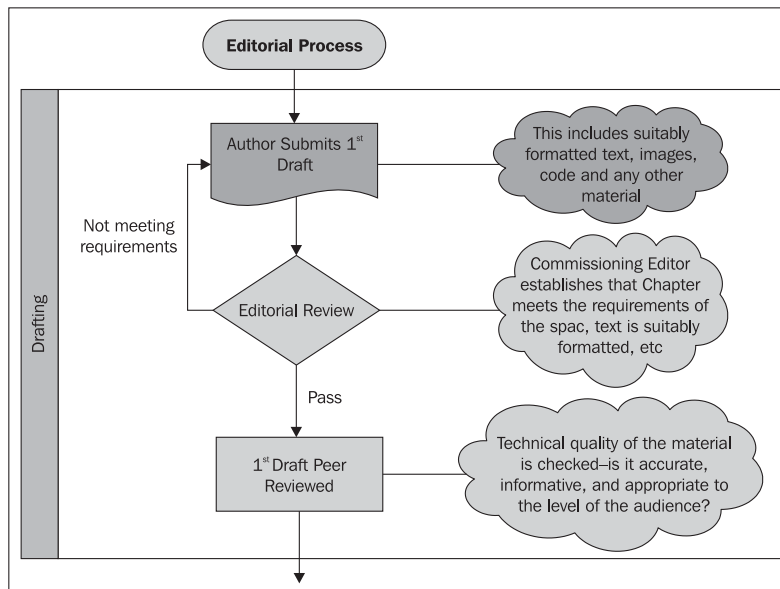
For More Information:
www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book

Connectivity API

All of the above sections were to get you used to the object model a bit, so that you can understand how to traverse a structured diagram and retrieve the information that you want. The **Connectivity API** also provides easy methods for creating and deleting connections, but we are simply interested in traversing connections in order to check or export the process steps to another application.

Here is the top part of my **Write Chapter Sub-process** page which demonstrates some of the key features of the Connectivity API. They are done in the following sequence:

1. The flow shapes are connected together creating a logical sequence of steps.
2. Some steps have an associated callout with extra Notes.
3. Some steps are within a Container shape to define the Phase.



Now we will traverse the diagram in code, and list out the steps in their phases with any associated notes, but first we need to understand a few of the new methods in the Connectivity API.

The Shape.ConnectedShapes method

The `Shape.ConnectedShapes` method returns an array of **identifiers (IDs)** of shapes that are one degree of separation away from the given shape (that is, separated by a 1-D connector).

The method has two arguments, `Flags` and `CategoryFilter`.

- `Flags`: This filters the list of returned shape IDs by the directionality of the connectors, using the `VisConnectedShapesFlags` enum for `All`, `Incoming`, or `Outgoing` nodes.
- `CategoryFilter`: This filters the list of returned shape IDs by limiting it to IDs of shapes that match the specified category. A shape's categories can be found in the `User.msvShapeCategories` cell of its `ShapeSheet`.

So, we can use the new `ConnectedShapes` method to list all of the significant connections in my **Write Chapter Sub-process** page. I have used the existence of the `Prop.Cost` cell as a test for shape significance.

```
Public Sub ListNextConnections()  
    Dim shp As Visio.Shape  
    Dim connectorShape As Visio.Shape  
    Dim sourceShape As Visio.Shape  
    Dim targetShape As Visio.Shape  
    Dim aryTargetIDs() As Long  
    Dim arySourceIDs() As Long  
    Dim targetID As Long  
    Dim sourceID As Long  
    Dim i As Integer  
    Const CheckProp As String = "Prop.Cost"  
    For Each shp In Visio.ActivePage.Shapes  
        If Not shp.OneD Then  
            If shp.CellExists(CheckProp, Visio.visExistsAnywhere) Then  
                Debug.Print "Shape", shp.Name, shp.Text  
                arySourceIDs =  
                    shp.ConnectedShapes(visConnectedShapesOutgoingNodes, "")  
                For i = 0 To UBound(arySourceIDs)  
                    Set sourceShape =  
                        Visio.ActivePage.Shapes.ItemFromID(arySourceIDs(i))  
                    If sourceShape.CellExists(CheckProp,  
                        Visio.visExistsAnywhere) Then  
                        Debug.Print , "<", sourceShape.Name,  
                            sourceShape.Text  
                    End If  
                Next  
                aryTargetIDs =  
                    shp.ConnectedShapes(visConnectedShapesIncomingNodes, "")  
                For i = 0 To UBound(aryTargetIDs)  
                    Set targetShape =  
                        Visio.ActivePage.Shapes.ItemFromID(aryTargetIDs(i))  
                    If targetShape.CellExists(CheckProp,
```

```

        Visio.visExistsAnywhere) Then
            Debug.Print , ">", targetShape.Name,
                targetShape.Text
        End If
    Next
End If
End If
Next
End Sub

```

The top of the output from this function will appear as follows:

Shape	Start/End	Editorial Process	
	<	Document	Author Submits 1st Draft
Shape	Document	Author Submits 1st Draft	
	<	Decision	Editorial Review
	>	Start/End	Editorial Process
	>	Decision	Editorial Review
Shape	Decision	Editorial Review	
	<	Document	Author Submits 1st Draft
	<	Process	1st Draft Peer Reviewed
	>	Document	Author Submits 1st Draft

The Shape.GluedShapes method

The `Shape.GluedShapes` method returns an array of identifiers for the shapes that are glued to a shape. For instance, if the given shape is a 2-D shape that has multiple connectors attached to it, this method would return the IDs of those connectors. If the given shape is a connector, this method would return the IDs of the shapes to which its ends are glued.

The method has three arguments, `Flags`, `CategoryFilter`, and `OtherConnectedShape`:

- `Flags`: This filters the list of returned shape IDs by the directionality of the connectors, using the `VisGluedShapesFlags` enum for `All1D`, `All2D`, `Incoming1D`, `Incoming2D`, `Outgoing1D`, or `Outgoing2D` nodes.
- `CategoryFilter`: This filters the list of returned shape IDs by limiting it to IDs of shapes that match the specified category. A shape's categories can be found in the `User.msvShapeCategories` cell of its **ShapeSheet**.
- `OtherConnectedShape`: Optional additional shape to which returned shapes must also be glued.

The method is used as follows :

```
arIDs = Shape.GluedShapes(Flags, CategoryFilter,  
    pOtherConnectedShape)
```

The Shape.MemberOfContainers property

We can return an array of IDs of the **Containers** that a shape is within.

You can use the ID to return the Container shape, get its ContainerProperties object, and, in this case, return the text from the shape.

Here is a private function that I will use in the main function in the following code:

```
Private Function getContainerText(ByVal shp As Visio.Shape) As String  
    'Return text of any containers,  
    'or an empty string if there are none  
    Dim aryTargetIDs() As Long  
    Dim targetShape As Visio.Shape  
    Dim returnText As String  
    Dim i As Integer  
    returnText = ""  
    aryTargetIDs = shp.MemberOfContainers  
    On Error GoTo exitHere  
    For i = 0 To UBound(aryTargetIDs)  
        Set targetShape =  
            shp.ContainingPage.Shapes.ItemFromID(aryTargetIDs(i))  
        If Len(returnText) = 0 Then  
            returnText = targetShape.ContainerProperties.Shape.Text  
        Else  
            returnText = returnText & vbCrLf &  
                targetShape.ContainerProperties.Shape.Text  
        End If  
    Next  
  
    exitHere:  
    getContainerText = returnText  
End Function
```

The Shape.CalloutsAssociated property

This property will return an array of shape IDs of any associated callouts.

You can use the ID to return the callout shape, and, in this case, return the text from within that shape.

Here is a private function that I will use in the main function:

```
Private Function getCalloutText(ByVal shp As Visio.Shape) As String
'Return text of any connected callouts,
'or an empty string if there are none
Dim aryTargetIDs() As Long
Dim targetShape As Visio.Shape
Dim returnText As String
Dim i As Integer
    returnText = ""
    aryTargetIDs = shp.CalloutsAssociated
    On Error GoTo exitHere
    For i = 0 To UBound(aryTargetIDs)
        Set targetShape =
            shp.ContainingPage.Shapes.ItemFromID(aryTargetIDs(i))
        If Len(returnText) = 0 Then
            returnText = targetShape.Characters.Text
        Else
            returnText = returnText & vbCrLf &
                targetShape.Characters.Text
        End If
    Next
exitHere:
    getCalloutText = returnText
End Function
```

Listing the steps in a process flow

In order to create a sequential listing of the steps in the page, we need to create a function that will call itself to iterate through the connections out from the source shape.

```
Private Function getNextConnected(ByVal shp As Visio.Shape, ByVal
dicFlowShapes As Dictionary, ByVal colSteps As Collection) As
Collection
'Return a collection of the next connected steps
Dim aryTargetIDs() As Long
Dim targetShape As Visio.Shape
Dim returnCollection As Collection
Dim i As Integer
    dicFlowShapes.Add shp.NameID, shp

    aryTargetIDs =
        shp.ConnectedShapes(visConnectedShapesOutgoingNodes, "")
    For i = 0 To UBound(aryTargetIDs)
```

```
Set targetShape =  
    Visio.ActivePage.Shapes.ItemFromID(aryTargetIDs(i))  
If Not targetShape.Master Is Nothing And  
    dicFlowShapes.Exists(targetShape.NameID) = False Then  
    colSteps.Add targetShape  
    getNextConnected targetShape, dicFlowShapes, colSteps  
End If  
Next  
Set getNextConnected = colSteps  
End Function
```

Finally, we can create the public function that will list the steps. For simplicity, I'm only following the direct route and not displaying the text on the connector lines.

I have introduced the `Visio.Selection` object because it contains a collection of shapes returned by the `Page.CreateSelection()` method, which is extremely useful for getting a filtered collection of shapes by **Layer**, **Master**, **Type**, and so on.



I am also using the Dictionary object in the following code, so you will need to ensure that the **Microsoft Scripting Runtime** library (C:\Windows\system32\scrn.dll) is ticked in the **References** dialog opened from the **Tools** menu in the Visual Basic user interface.

```
Public Sub ListProcessSteps()  
Dim sel As Visio.Selection  
Dim pag As Visio.Page  
Dim shp As Visio.Shape  
Dim shpStart As Visio.Shape  
Dim shpEnd As Visio.Shape  
Dim iStep As Integer  
Dim dicFlowShapes As Dictionary  
Set dicFlowShapes = New Dictionary  
Set pag = Visio.ActivePage  
'Find the Start and End shapes on the Page  
'Assume that they are the instances of the Master "Start/End"  
'Assume that the Start has no incoming connections  
'and the End shape has no outgoing connections  
Set sel = pag.CreateSelection(visSelTypeByMaster, 0,  
    pag.Document.Masters("Start/End"))  
If Not sel.Count = 2 Then  
    MsgBox "There must be one Start shape and one End shape  
        only", vbExclamation, "ListProcessSteps"  
Exit Sub  
End If
```

```
For Each shp In sel
    If shpStart Is Nothing Then
        Set shpStart = shp
        Set shpEnd = shp
    Else
        If UBound(shp.ConnectedShapes(
            visConnectedShapesOutgoingNodes, "")) > -1 _
            And UBound(shp.ConnectedShapes(
                visConnectedShapesIncomingNodes, "")) = -1 Then
            Set shpStart = shp
        ElseIf UBound(shp.ConnectedShapes(
            visConnectedShapesIncomingNodes, "")) > -1 _
            And UBound(shp.ConnectedShapes(
                visConnectedShapesOutgoingNodes, "")) = -1 Then
            Set shpEnd = shp
        End If
    End If
Next
iStep = 1

Dim nextSteps As Collection
Dim nextShp As Visio.Shape
Dim iNext As Integer
Set nextSteps = New Collection
Set nextSteps = getNextConnected(shpStart, dicFlowShapes,
    nextSteps)
Debug.Print "Step", "Master.Name", "Phase", "Text", "Notes"
Debug.Print iStep, shpStart.Master.Name,
    getContainerText(shpStart), shpStart.Text,
    getCalloutText(shpStart)
For iNext = 1 To nextSteps.Count
    iStep = iNext + 1
    Set nextShp = nextSteps.Item(iNext)
    Debug.Print iStep, nextShp.Master.Name,
        getContainerText(nextShp), nextShp.Characters.Text,
        getCalloutText(nextShp)
Next
If Not nextShp Is shpEnd Then
    MsgBox "The process did not finish on the End shape",
        vbExclamation, "ListProcessSteps"
End If
End Sub
```


With a fanfare of trumpets, we get a simple listing of each step in order:

Step	Master.Name	Phase	Text	Notes
1	Start/End	Editorial Process		
2	Document	Drafting	Author Submits 1st Draft	This includes suitably formatted text, images, code and any other material
3	Decision	Drafting	Editorial Review	Commissioning Editor establishes that Chapter meets the requirements of the spec, text is suitably formatted, etc
4	Process	Drafting	1st Draft Peer Reviewed	Technical quality of the material is checked - is it accurate, informative, and appropriate to the level of the audience?
5	Process	Editing	Editorial Acceptance Verdict	Commissioning Editor evaluates reviewer comments to verify that the Chapter meets the "Editorial Acceptance" standard
6	Process	Editing	Author Rewrite	Author addresses comments, adds any extra material requested
7	Process	Editing	Final Edit	
8	Decision	Editing	Pass?	Finer iterations of chapter required?
9	Process	Production	Production Phase	Indexing, Layout, Proofing
10	Process	Production	Author Review of "PreFinal" PDF	Author inspects finished PDF to see if there are any last minute changes required and if they are happy with the chapters
11	Start/End	Publication		

Summary

In this chapter, we delved into the Visio object model, and looked at the hierarchy of the objects and collections.

We looked at the analytical parts of the Connectivity API, which enabled us to navigate connections and to retrieve surrounding containers and associated callouts.

We also used this knowledge to build a function that does some rudimentary checks of a diagram structure, and to list the steps in a process flow.

In the next chapter, we will look into the ShapeSheet and how to use the functions within it.

Where to buy this book

You can buy Microsoft Visio 2010 Business Process Diagramming and Validation from the Packt Publishing website: <https://www.packtpub.com/microsoft-visio-2010-business-process-diagramming/book>

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.PacktPub.com/microsoft-visio-2010-business-process-diagramming/book