PROSPECTOR™

The Gold Standard In Toolmaking

User Guide to the Prospector Object Model

January 2001

Summary: The Prospector object model allows you to customize Prospector to your specific needs. This guide includes examples and exercises on how to develop macros using the model, find the right tools in the model to solve a given problem, understand key programming concepts, and gain a working knowledge of the Microsoft Visual Basic® programming language.

Contents

Introduction

Customizing Prospector with the Prospector object model is easy. You don't need to know C or C++ or any programming language. You don't need to know anything about object models. You don't even have to know how to program your VCR.

Prospector uses VBScript as it's scripting engine. VBScript is a subset of Visual Basic. Solutions you create in VBScript are called *macros*. A macro is a series of instructions in VBScript that perform something useful. When you write macros to perform tasks in Prospector, you write VBScript instructions that use the Prospector object model.

Why Should I Learn How to Use Prospector Object Model?

The Prospector object model allows you to customize Prospector to suit your organization's specific needs. The Prospector object model is great when someone in your organization demands (or if they're polite, requests) additional functionality in Prospector.

Using This Guide

This guide is organized into four lessons. Each lesson is hands-on—you will use the lessons with Prospector as you go. The guide is best used not as bedtime reading but as be-in-front-of-your-computer-and-try-it reading.

What You Need to Know Before You Get Started

All you need is a working knowledge of Microsoft Windows and a familiarity with Prospector.

What You'll Know When You Finish the Lessons

After going through this guide and doing all the examples and exercises, you will be able to develop applications using the Prospector object model. You will also be able to discover on your own how to find the right tools in the Prospector object model to solve a given problem. You will understand some key programming concepts and gain a working knowledge of VBScript.

In a nutshell, with the help of this guide, you will be able to apply the Prospector object model to meet your organization's needs.

Setting Up

To use this guide and go through the included examples, you need to have Prospector version 5.0 or later. Because this guide uses VBScript, a new feature in Prospector 5.0, the examples and lessons will not work with earlier versions of Prospector.

You do not need any special development tools to use the Prospector object model.

Lesson 1: Getting Started

You'll need to know how to load macro files and run macros in Prospector. To load a macro file:

1. Start Prospector, if it is not already running.
2. In the **Tools** menu, click on **Customize.** This will bring up the customization dialog.
3. Click on the **Macro Files** tab. This tab contains a list of macro files that are available to Prospector.
4. Click on the **Add** button. This will bring up a standard file browser that you can use to find and select your macro file. Navigate your way over to where Prospector is installed (usually C:\Program Files\Prospector). In the **Examples** directory is a sample macro file called **Macros.pvb.** Select the file and click on the **Open** button.

5. Now that the macro file is in the list, check the box next to the name to enable the macro file. Prospector only loads macro files that are enabled.

6. Hit the **OK** button. The macro file is now loaded and the macros are available to run.

Let's try one of the macros.

1. In Prospector, load the sample project 1500/Ejector, or any project that has surface data.

2. In the **Tools** menu, click on **Macro**. This brings up a dialog that lets you run macros and scripts.

3. Click on the **Macros** tab. This tab displays a list of macros that are available to Prospector. The macros are from the file specified in the **Macro File** combo box. Since we only have one macro file, the **Macros.pvb** file is already selected and the list contains the macros in that file.

4. In the list, click on **SetSurfaceColorBlue** and then click the **Run** button. The dialog closes and now all the surfaces in the model are blue.

Like other settings, you only need to add and enable the macro file once. The next time you start Prospector, the macro file will loaded automatically. Also, Prospector will keep the macro file in sync with what is loaded. When you make a change to the macro file (by changing a macro or adding a macro) Prospector knows that it changed. The next time you try to run one of the macros, Prospector will prompt you as to whether or not it should reload the macro file.

You can assign macros to keys, strokes, or toolbar buttons. See the Help for Prospector to learn how to do this.

Lesson 2: Programming Concepts: Sub…End Sub and Procedures

So now that you've gotten your feet wet in this stuff, it's time to learn a bit about the water you're standing in. Let's take a close look at the macro you just ran. Load the **Macros.pvb** into your favorite text editor, such as Notepad. Here's the macro **SetSurfaceColorBlue**:

```
Sub SetSurfaceColorBlue()
    ' DESCRIPTION: Set the surfaces to be color blue
    SetSurfaceColor gmColorDarkBlue
End Sub
```

Let's first look at the VBScript key words **Sub** and **End Sub**. **Sub…End Sub** are used to begin and end a macro, following the pattern shown below:

```
Sub AnyNameHere()
  Some of that cool refreshing Object Model code here
End Sub
```

AnyNameHere is the name of a macro or *procedure*. A procedure is a small set of code that you create that does something. **SetSurfaceColorBlue** is an example of a procedure. A procedure doesn't have to be a macro, however. You can create a procedure and then "call" that same procedure from another procedure. **SetSurfaceColorBlue** calls another procedure **SetSurfaceColor**:

```
Private Sub SetSurfaceColor(nColor)
    ' DESCRIPTION: Set the surface color to the specified value
    Dim SurfaceList
    Set SurfaceList =
ActiveProject.Model.GeomEntityCollection.Restrict("[Type] And
gmTypeAllSurfaces")
    If Not SurfaceList Is Nothing Then
        SurfaceList.SetColor nColor
    End If
End Sub
```

If you run **SetSurfaceColorBlue**, the procedure **SetSurfaceColor** will run. The keyword, **Private**, means that the procedure cannot be used by other modules, or macro files. Also, private macros do not show up in any macro list in Prospector. For example, in the first lesson, the macro list in the **Macros** tab did not show **SetSurfaceColor**, but did show **SetSurfaceColorBlue**.

So why would you want to do this? Creating separate procedures allows you to organize your code in a nice, clean way, and it allows you to do common procedures easily. For example, two other macros, **SetSurfaceColorGreen** and **SetSurfaceColorRed** also call **SetSurfaceColor**.

Sometimes procedures need additional information. The macro **SetSurfaceColor** needs to know which color to apply to the surfaces. The **SetSurfaceColorBlue** calls **SetSurfaceColor** with one parameter, **gmColorBlue**.

Lesson 3: Programming Concepts: Objects, Methods, and Properties

At some point, you may have heard all the hoopla over "object-oriented" programming. Object-oriented programming is the key concept behind C++ and Java, the most widely used programming languages today. What you probably didn't know, however, is that just by finishing Lessons 1 and 2, you can now call yourself an object-oriented programmer!

That's right. The Prospector object model uses object-oriented programming. Fortunately, to use and understand the Prospector object model, you don't need to take a class in the subject or write a thesis on it. To gain a working knowledge of the Prospector object model, you only need to know three concepts:

Concept	Description	Example
Object	A "thing"	Profile
Method	Something a "thing" can do	Reverse direction

Property	A characteristic of a "thing"	Number of control points

Everyday things can be thought of as objects, methods, and properties. For instance, consider a car as an object. A car object has methods, which are various things it can do, such as Drive, Start, Turn Left, Turn Right. A car also has properties that describe it: the color is beige, and the number of headlights is two. Properties can be VBScript types such as the **number** of headlights. Properties can also be defined types, such as the **color** of the car. Properties can also be objects. For example, a car has a property called engine. The engine is an object that has it's own methods and properties.

Take a closer look **SetSurfaceColor** and see where the objects, methods, and properties are:

```
Set SurfaceList =
ActiveProject.Model.GeomEntityCollection.Restrict("[Type] And
gmTypeAllSurfaces")
If Not SurfaceList Is Nothing Then
    SurfaceList.SetColor nColor
End If
```

There are several objects in this code. The first object is the **ActiveProject** object. This object is actually a property of the **Application** object. Since the **Application** object is the top-level object, you don't need to specify **Application.ActiveProject,** you can just specify **ActiveProject.** This object is a **Project** object. The **Project** object has a property called **Model** that is a **Model** object. The **Model** object has a property called **GeomEntityCollection** that is a **GeomEntityCollection** object, which is a list of **GeomEntity**'s. To use a property, you simply place a period between the object and the property. For example, `ActiveProject.Model.`

Whenever you first use an object, you have to use the **Set** key word. Objects take up memory in the computer; the **Set** key word allocates the memory required for an object.

You can give objects any name you want. In the above example, I gave the surface list object the name, **SurfaceList,** but you can change the name to suit your mood.

The **SetSurfaceColor** code contains two methods: the first is **Restrict**, and the second is **SetColor**. A method is always associated with an object. In this case, **Restrict** is associated with the object, **GeomEntityCollection**. To use a method, you simply place a period in between the object and the method. For example, `List.SetColor`.

Sometimes methods need additional information. For example, the **SetColor** method needs to know what color to apply to all the entities in the list. We tell it to use the color specified as a parameter just like passing in a parameter to the macro. Some methods require more than one piece of information, while others require none.

One last thing to know: Every object is of a specific type. Each type of object has its own set of methods and properties. In the above example, **ActiveProject** is a "project" object. Projects have methods, such as **Print**, and properties, such as **JobNumber**, that other types of objects do not have. For example, the following instruction:

```
Model.JobNumber = "1500"
```

would not work because objects of type "Model" do not have properties called **JobNumber**.

Lesson 4: Real-World Example #1

Suppose that you want to change the number of flow lines displayed for each surface. Using the macros in the previous lessons, let's consider how you would do this as a user in Prospector. Sometime you may want to see 2 flow lines, other times you may want to see 3 or 4. So, similar to the **SetSurfaceColor** macro, we'll make a private macro that has a parameter, the number of flow lines. In the macro file, copy the **SetSurfaceColor** macro, change the macro name and parameter name. The rest we will reuse, since it is getting a list of surfaces. The macro should look like this:

```
Private Sub SetSurfaceFlowCount(nCount)
    ' DESCRIPTION: Set the surface color to the specified value
    Dim SurfaceList
    Set SurfaceList =
ActiveProject.Model.GeomEntityCollection.Restrict("[Type] And
gmTypeAllSurfaces")
    If Not SurfaceList Is Nothing Then
        SurfaceList.???????????????
    End If
End Sub
```

Now we hope that the **GeomEntityCollection** object has a method that changes the number of flow lines. It does: **SetFlowLineCount**. So the line in the middle of the 'If" clause needs to be:

```
SurfaceList.SetFlowLineCount nCount
```

Now that we have the main procedure done, we can write any number of macros that call **SetSurfaceFlowLineCount**. For now, let's just write three:

```
Sub SetSurfaceFlowLineCount2()
    ' DESCRIPTION: Set the flow line count to 2
    SetSurfaceFlowLineCount 2
End Sub

Sub SetSurfaceFlowLineCount3()
    ' DESCRIPTION: Set the flow line count to 3
    SetSurfaceFlowLineCount 3
End Sub

Sub SetSurfaceFlowLineCount4()
    ' DESCRIPTION: Set the flow line count to 4
    SetSurfaceFlowLineCount 4
End Sub
```

Now let's run one of the new macros. In Prospector, under the **Tools** menu, select the **Macro** item. Since we changed the macro file, Prospector asks if we want to reload it. Click on the **Yes** button, and the new macros appear in the list. Select the macro **SetSurfaceFlowLineCount2** and click on the **Run** button. The surfaces now have two flow lines.

Extra Credit

Hook up the flow line macros to the keys 2, 3, and 4.

Lesson 5: Programming Concept: If This, Then That

Sometimes we need to control which code is used, based on the state of things. We can create procedures that respond to different conditions by using the **If...Then** control statement. The **If...Then** control statement is one of many VBScript tools that can direct the flow of your code. The format of the **If...Then** control statement is as follows:

```
If <condition> Then
<code here>
End If
```

In the above code, <condition> represents something that can be True or False. <code here> represents the code that will run if <condition> is determined to be True.

<condition> examples:

Profile.Layer = 3	True if the profile is on layer 3.
List.Count > 0	True if the number of entities in the list is greater than 0.
Profile.LineStyle >= 2 And Profile.LinStyle <= 5	True if the profile's line style is between 2 and 5.

You can also check for valid objects. In the **SetSurfaceColor** macro, the **If..Then** control statement checks to make sure the **Restrict** method returned a valid list:

```
If Not SurfaceList Is Nothing Then
    SurfaceList.SetColor nColor
End If
```

If the **Restrict** method returns us a null list, we will know not to call methods or look at properties of the list.

You can handle multiple conditions by using the **ElseIf** clause as well as handle the condition when no other conditions are met:

```
If Profile.Layer = 0 Then
    ' Do something if the profile is on layer 0
ElseIf Profile.Layer = 1 Then
    ' Do something if the profile is on layer 1
Else
    ' Do something if the profile is on any other layer
End If
```

Lesson 6: Programming Concept: Collections

So far, you have learned about objects, properties, methods, and events. You need to understand one last category to fully use the Prospector object model: *Collections*.

A collection is a special type of object—an object that is a group of other objects. So, for example, if "car" is an object, "cars" is a collection of cars.

A collection can also be a property of another object. Continuing with the car example, "doors" can be a property of a "car" object as well as a collection of "door" objects. Therefore, we can understand the relationship like this:

"Cars" is a collection of "car" objects. Each "car" object has a property called "doors." "Doors" is a collection of "door" objects.

Working With Collections

All collections have methods and properties that allow you to access the individual objects in the collections. The **Count** property is the number of objects in the collection. A collection can be empty, in which case the **Count** property would return 0. The **Item** method returns a specific object in a collection based on an index. Prospector collections are 0-based, so the first item is indexed at 0, the second is 1, etc. For example,

```
Set Setup2 = ActiveProject.Setups.Item(1)
```

The object **setup2** will become the second **Setup** in the **Setups** collection of the active project.

You can use the **Count** property and **Item** method to loop through a collection like so:

```
For I = 0 To List.Count - 1
    Set Object = List.Item(I)
    ' Do something with the Object
Next I
```

However, there is an easier way, using the **For Each…Next** control statement:

```
For Each Object In List
    ' Do something with the Object
Next I
```

These will do exactly the same thing.

GeomEntityCollection

When working with geometry, the most important collection is the **GeomEntityCollection**. We've seen that the **Model** object has a collection property also called **GeomEntityCollection.** When working with this collection, you use the same properties and methods as described above. However, there are additional methods that allow you to affect all the entities in the list without having to traverse the list. In the example macros, we've already have seen two methods, **SetColor** and **SetFlowLineCount**.

One other very useful method is the **Restrict** method, which was also used in the example macros. This method returns another **GeomEntityList**, which contains a subset of the original list, "restricted" to the filter described in the string. The filter string can contain any number of restrictions. The restrictions are based on the properties available to the **GeomEntity** object, since these are what are contained in the collection. Some properties of the **GeomEntity** object are **Color, Layer, LineStyle,** and **Type**. The restrictions are like the condition clauses of the **If..Then** control statement, with the property surround by brackets [].

Here are some examples of the **Restrict** method:

```
Set List = OtherList.Restrict("[Color] = gmColorBlue")
Set List = OtherList.Restrict("[Layer] >= 2 And [Layer] <= 5")
Set List = OtherList.Restrict("[Color] = gmColorBlue And [Layer] = 7")
```

The first example gets a list of all blue entities. The second example gets all entities on layers 2 through 5. The third example gets all blue entities on layer 7.

The **Type** property is different in that you don't want to use numeric operators like =, <=, <>, etc. Instead use the bit wise operators **And** and **Not**. Surface types are:

```
gmTypeSurface          a regular surface
gmTypeOffsetSurface    an offset surface
gmTypeTrimSurface      a trimmed surface
```

The constant, **gmTypeAllSurfaces** is a combination of all these. So if we want to get a list of all of the surfaces, like we did in **SetSurfaceColor**, we would use this line:

```
Set SurfaceList =
ActiveProject.Model.GeomEntityCollection.Restrict("[Type] And
gmTypeAllSurfaces")
```

Check the Prospector object model Reference Guide for more information on types and properties of geometric entities.

Lesson 7: Real-World Example #2

So we've got surfaces drawn in the right color (**SetSurfaceColor**) with the right number of flow lines (**SetSurfaceFlowLineCount**). But now we want to be able to hide and unhide surfaces and profiles. It sound like we can make one private macro that takes two parameters, type (surfaces or profiles) and whether to hide or unhide. The method to hide or unhide is called **SetShow**. This method takes one parameter, either **True** or **False**. If **True** the method will unhide the entities; if **False** the method will hide the entities. Here's the macro:

```
Private Sub HideOrUnhideByType(nType, bShow)
    ' DESCRIPTION: Hide or unhide specified class
    Dim ClassList
    Set ClassList =
ActiveProject.Model.GeomEntityCollection.Restrict("[Type] And " & nType)
    If Not ClassList Is Nothing Then
        ClassList.SetShow bShow
    End If
End Sub
```

Now we can easily make the macros we want to run:

```
Sub HideSurfaces()
    ' DESCRIPTION: Hide all surfaces
    HideOrUnhideByType gmTypeAllSurfaces, False
End Sub

Sub UnhideSurfaces()
    ' DESCRIPTION: Unhide all surfaces
    HideOrUnhideByType gmTypeAllSurfaces, True
End Sub
```

```
Sub HideProfiles()
    ' DESCRIPTION: Hide all profiles
    HideOrUnhideByType gmTypeAllProfiles, False
End Sub

Sub UnhideProfiles()
    ' DESCRIPTION: Unhide all profiles
    HideOrUnhideByType gmTypeAllProfiles, True
End Sub
```

Note that **gmTypeAllProfiles** is like **gmTypeAllSurfaces**. There are two types of profiles: **gmTypePointProfile** which means the profile is made of only points (used for drilling operations) and **gmTypeProfile** which means the profile is made of any geometry (lines, arcs, splines). So the type **gmTypeAllProfiles** encompasses both types. Let's add one more macro that will unhide everything:

```
Sub UnhideAll()
    ' DESCRIPTION: Unhide all entities
    ActiveProject.Model.GeomEntityCollection.SetShow True
End Sub
```

Since the **GeomEntityCollection** property of the project's model is a geometry entity collection, we can call the methods directly on that object. No need to call the **Restrict** method.

Where to Go from Here

Now you have been introduced to the Prospector object model. You can now create custom Prospector macros.

All the macros developed in this guide are in the **Macros.Complete** file in the **Examples** directory.

Of course, this guide only scratched the surface of all the powerful things the Prospector object model can do. Use the online Help to explore the many collections, objects, methods, and events at your disposal.

Congratulations on becoming a real Prospector object model programmer! Now you can start using the Prospector object model to save your company time and money!