Are you WMI or Not (Part II)

by Craig Murphy

In the last issue of the Developer's Magazine, Craig provided an introduction to Microsoft® Windows® Management Instrumentation (WMI) and demonstrated how it could be used to make laborious tasks a little easier. In this follow-up article, Craig demonstrates how WMI can be used in Delphi and how it can be used to retrieve information from remote computers.

Introduction

I was a little bit embarrassed at the lack of any Delphi code in part one of this article. Worse still, I included some VBScript and came close to ranting on about how useful it could be. However, nobody flamed me, nobody added me to any e-mail spam lists that I've not managed to get myself onto already, and nobody e-mailed our esteemed editor to complain (at least I haven't about any complaints!). Hopefully the VBScript didn't just make you turn the page to start reading about Media Player Visualisations – which you read after reading part one of this article, right? (I'll be testing you on both articles later on…only kidding!)

Anyway, this article is the result of my previously mentioned embarrassment – it is 100% Delphi. Over the course of this article I will create a WMI test application that lets us use WMI in our Delphi applications. I'll even demonstrate how Delphi can replicate VB and VBScript's *for each* construct.

First Things First

As you might imagine, using WMI in a Delphi application requires us to import a type library. Importing a type library is easy – from Delphi's Project menu, select the Import Type Library menu option. Figure 1 identifies the type library that we need to import.

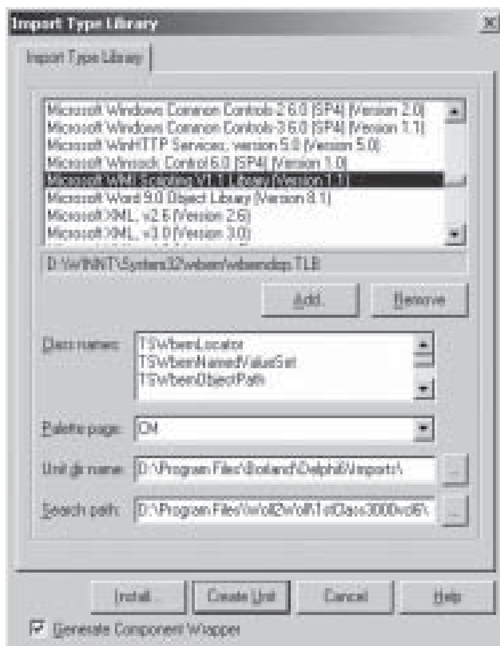

Figure 1 – Importing the WMI type library

Re-cap

The WMI Scripting type library will be used to build a number of small applications. The first application will essentially be a Delphi version of the Visual Basic Script (VBScript) example that I wrote about in part one of this article. For the sake of completeness, Listing 1 presents the VBScript that we will attempt to replicate. Essentially we are going to write some code that extracts information from the Win32_OperatingSystem WMI class.

```
' Listing1.vbs
' demonstrates extractions of OS info
' using WMI

strComputer = "."

' use the default namespace
Set objWMIService = _
  GetObject("winmgmts:\\" & strComputer)

' use the specified namespace
'Set objWMIService = GetObject( _
'"winmgmts:\\" & strComputer & "\root\cimv2")

Set colOperatingSystems =    _
  objWMIService.InstancesOf( _
  "Win32_OperatingSystem")

For Each objOpSys In colOperatingSystems
  Wscript.Echo _
   "Name:          " _
    & objOpSys.Name            & vbCrLf & _
   "Caption:       " _
    & objOpSys.Caption         & vbCrLf & _
   "CurrentTimeZone:" _
    & objOpSys.CurrentTimeZone & vbCrLf & _
   "LastBootUpTime: " _
    & objOpSys.LastBootUpTime  & vbCrLf & _
   "LocalDateTime:  " _
    & objOpSys.LocalDateTime   & vbCrLf & _
   "Locale:        " _
    & objOpSys.Locale          & vbCrLf & _
   "Manufacturer:   " _
    & objOpSys.Manufacturer    & vbCrLf & _
   "OSType:        " _
    & objOpSys.OSType          & vbCrLf & _
   "Version:       " _
    & objOpSys.Version         & vbCrLf & _
   "Service Pack:   " _
    & objOpSys.ServicePackMajorVersion  & _
   "." & objOpSys.ServicePackMinorVersion _
    & vbCrLf &                            _

   "Windows Directory: " _
    & objOpSys.WindowsDirectory
Next
```

Listing 1 – Using WMI in VBScript

It's Not So Easy

Implementing WMI in a compiled environment such as Delphi requires a greater understanding of the WMI architecture. VBScript is an interpreted scripting language, whereas Delphi is a compiled language. Interpreted languages are good in environments where late binding is prevalent. In the VBScript example (listing 1), how does the object objOperatingSystem know about the properties Caption, CurrentTimeZone, LastBootUpTime, etc? Well, it does not know anything about those properties until the objWMIService.InstancesOf(..) method has been executed. In other words, knowledge of the properties becomes evident at run-time. This is fine in an interpreted environment, however it does present us with some problems in a compiled environment – after all, the Delphi compiler would need to know about the properties at design time.

Just how we handle this issue of early-binding vs late-binding will be discovered later in this article.

Getting Started

The WMI Scripting type library exposes one object that is of use to us: TSWbemLocator. On its own, it's not much use – we need to connect something. It is an instance of this object that will allow us to connect to the local WMI server. Listing 2 demonstrates how to create an instance of TSWbemLocator that connects to the local WMI server.

```
var
  wmiLocator:   TSWbemLocator;
  wmiServices:  SWbemServices;
...

wmiLocator  := TSWbemLocator.Create(self);
wmiServices := wmiLocator.ConnectServer(
                 '.',
                 'root\cimv2',
                 '', '', '', '', 0, nil);
```

Listing 2 – Connecting to local WMI server

Listing 3 presents the ConnectServer method interface. We specified '.' for the strServer parameter, indicating that we are connecting to the local computer. The strNamespace parameter took the value 'root\cimv2'. The remaining parameters, strUser, strPassword, etc. were ignored. As you might imagine, it is this function that will be used to connect to remote computers too, but about that later in this article.

The ConnectServer method returns an object that adheres to the ISWbemServices interface – this is used to perform actions against the chosen computer and namespace. An example of an action might be to obtain the computer's operating system information, as in our example.

```
function ConnectServer(
  const strServer: WideString;
  const strNamespace: WideString;
  const strUser: WideString;
  const strPassword: WideString;
  const strLocale: WideString;
  const strAuthority: WideString;
  iSecurityFlags: Integer;
  const objWbemNamedValueSet: IDispatch):
    ISWbemServices; safecall;
```

Listing 3 – ConnectServer Interface

this code is on www.richplum.co.uk/html/downloads.asp

ISWbemServices provides us with a means of extracting the WMI information for a given WMI class – in our case it is Win32_OperatingSystem. The InstancesOf method is used to obtain an object that adheres to the ISWbemObjectSet interface. Listing 4a presents the snippet of code that would be used to obtain a collection of WMI objects that relate to the Win32_OperatingSystem WMI class (as we will see later, this is actually a collection consisting of just one WMI object).

```
wmiObjectSet := wmiServices.InstancesOf(
                  'Win32_OperatingSystem',
                  wbemFlagReturnImmediately or
                  wbemQueryFlagShallow, nil);
```

Listing 4a

Referring to listing 4a, wmiObjectSet is an object that adheres to the ISWbemObjectSet interface - it represents collections of these objects. Listing 4b presents the InstancesOf interface, highlighting the ISWbemObjectSet return interface type.

```
function InstancesOf(
  const strClass: WideString;
  iFlags: Integer;
  const objWbemNamedValueSet: IDispatch):
    ISWbemObjectSet; safecall;
```

Listing 4b – The InstancesOf Interface

The InstancesOf method provides us with access to a collection of objects that implement the ISwbemObject interface via the object it returns. Listing 5 presents the interface for a single ISWbemObject – keep in mind that we have a collection of these (zero or more) so, at some point in time, we will have to write some code that iterates over this collection.

```
ISWbemObject = interface(IDispatch)
  …
  property Qualifiers_: ISWbemQualifierSet
             read Get_Qualifiers_;
  property Properties_: ISWbemPropertySet
             read Get_Properties_;
  property Methods_: ISWbemMethodSet read
             Get_Methods_;
  property Derivation_: OleVariant read
             Get_Derivation_;
  property Path_: ISWbemObjectPath read
             Get_Path_;
  property Security_: ISWbemSecurity read
             Get_Security_;
end;
```

Listing 5 – The ISWbemObject interface

Each of the properties that we are interested in retrieving, Caption, CurrentTimeZone, LastBootUpTime, etc., are ultimately represented by the Properties_ property of the ISWbemObject. Each property will be held in the Properties_ collection. An object adhering to the ISWbemProperty interface, as shown in Listing 6, represents a single property.

```
ISWbemProperty = interface(IDispatch)
  ...
  property Name: WideString read Get_Name;
  property IsLocal: WordBool read Get_IsLocal;
  property Origin: WideString read Get_Origin;
  property CIMType: WbemCimtypeEnum read
             Get_CIMType;
  property Qualifiers_: ISWbemQualifierSet
             read Get_Qualifiers_;
  property IsArray: WordBool read Get_IsArray;
end;
```

Listing 6 – The WMI Property interface

Properties have names, such as Caption, CurrentTimeZone, LastBootUpTime, etc. They also have types, identified by the property CIMType. Listing 7 presents the possible values for CIMType – as you can see, they represent the data types that we are used to working with.

```
const
  wbemCimtypeSint8 = $00000010;
  wbemCimtypeUint8 = $00000011;
  wbemCimtypeSint16 = $00000002;
  wbemCimtypeUint16 = $00000012;
  wbemCimtypeSint32 = $00000003;
  wbemCimtypeUint32 = $00000013;
  wbemCimtypeSint64 = $00000014;
  wbemCimtypeUint64 = $00000015;
  wbemCimtypeReal32 = $00000004;
  wbemCimtypeReal64 = $00000005;
  wbemCimtypeBoolean = $0000000B;
  wbemCimtypeString = $00000008;
  wbemCimtypeDatetime = $00000065;
  wbemCimtypeReference = $00000066;
  wbemCimtypeChar16 = $00000067;
  wbemCimtypeObject = $0000000D;
```

Listing 7 – CIMTypes

Listings 2 through to 6 describe an element of hierarchy and also highlight two collections. Avoiding the urge to use elaborate diagramming tools, this is the hierarchy:

```
Locator
  Services
     ObjectSet (collection)
      Object
          Properties (collection)
```

The properties collection holds the information for each of the properties about which we are interested in retrieving information, Caption, CurrentTimeZone, LastBootUpTime, etc.

I did warn you that using WMI in Delphi was a little more complicated that using WMI in a scripting language! Before we start using everything that I have just covered, let us take a look at a simple example that uses some of the concepts.

A Simple Example

Now that I have set the scene, let us take a look at some code that actually does something useful. If you recall our original VBScript example where we extracted the basic operating system information, listing 8 presents the Delphi code that is required to come close to simulating the VBScript output.

```
procedure TfrmMain.btnPopupClick(Sender:
                      TObject);
var
  wmiLocator:   TSWbemLocator;
  wmiServices:  ISWbemServices;
  wmiObjectSet: ISWbemObjectSet;
  wmiObject:    ISWbemObject;
  Enum:         IEnumVariant;
  ovVar:        OleVariant;
  lwValue:      LongWord;
begin
  wmiLocator    := TSWbemLocator.Create(self);
  if edtPass.Text = '' then edtUser.Text := '';

  try
    wmiServices := wmiLocator.ConnectServer(
                      edtComputer.Text,
                      edtNamespace.Text,
                      edtUser.Text,
                      edtPass.Text,
                      '', '', 0, nil);

      // Obtain an instance of the WMI class
      wmiObjectSet := wmiServices.InstancesOf(
                     'Win32_OperatingSystem',
                      wbemFlagReturnImmediately or
                      wbemQueryFlagShallow, nil);

      // Replicate VBScript's "for each"
                       construct
      Enum := (wmiObjectSet._NewEnum) as
                       IEnumVariant;
      while (Enum.Next(1, ovVar, lwValue) =
                       S_OK) do begin
        wmiObject := IUnknown(ovVar) as
                       SWBemObject;
       Showmessage(wmiObject.GetObjectText_(0));
      end;
    finally
      wmiLocator.Free;
    end;
  end;
```

Listing 8 – WMI in Delphi

Listing 8 is part of an application that accompanies this article. I have attached listing 8 to a button's OnClick event – Figure 2 presents the message box that we would expect to see had we clicked on that button. As you can see it is similar to the VBScript output, however it is not built up using the individual properties (the VBScript used: objOperatingSystem.Caption, etc. to extract the properties). Instead we have a neatly formatted string that contains all the properties for the Win32_OperatingSystem WMI class. This could still prove to be useful; regular expressions could be used to extract the individual properties or the string could be e-mailed if it was required for support purposes.

Figure 2 – Simple WMI in Delphi

We are lucky with the Win32_OperatingSystem WMI class – it only returns a single WMI object that contains the properties we are looking for. There are other WMI classes that return more than one WMI object and it is for this reason that listing 8 has to replicate VBScript's *for each* language construct. So, whilst listing 1 and listing 8 contain code that iterates over a collection, neither of them iterates over more than one item because Win32_Operating does not contain any WMI subclasses. Indeed, the iFlags parameter of the InstancesOf method can be used to specify how the method should operate. For example, wbemQueryFlagShallow indicates that the collection that is returned should only contain the *immediate subclasses of the specified superclass*. Naturally, the converse, wbemQueryFlagDeep, instructs the InstancesOf method to traverse the WMI superclass recursively, returning all subclasses and their subclasses, etc.

Iterating over the WMI properties

Whilst listing 8 and figure 2 demonstrated that WMI could be used in our Delphi applications, it would still be useful if we could iterate over the individual properties extracting their values one by one. On the surface, this should be a fairly easy task – surely we just iterate over the **Properties_** that we saw in listing 5? The answer to this question is "yes, but….". Yes, but we have to identify each property's data type – the problem with some data types is that they themselves may contain collections (arrays). Nonetheless, let us go ahead and look at listing 9 – it presents an augmented version of listing 8; this time it iterates over the **Properties_** property of the WMI object that implements ISWbemObject.

```
procedure TfrmMain.btnGetWMIClick(Sender:
                         TObject);
var
  wmiLocator:   TSWbemLocator;
  wmiServices:  ISWbemServices;
  wmiObjectSet: ISWbemObjectSet;
  wmiObject:    ISWbemObject;
  propSet:      ISWbemPropertySet;
  wmiProp:      ISWbemProperty;
  propEnum,
  Enum:         IEnumVariant;
  ovVar:        OleVariant;
  Count:        Integer;
  lwValue:      LongWord;
  sValue:       String;
begin
  wmiLocator    := TSWbemLocator.Create(self);
  if edtPass.Text = '' then edtUser.Text := '';

  try
    wmiServices := wmiLocator.ConnectServer(
                      edtComputer.Text,
                      edtNamespace.Text,
                      edtUser.Text,
                      edtPass.Text,
                      '', '', 0, nil);

    // Obtain an instance of the WMI class
    wmiObjectSet := wmiServices.InstancesOf(
                   'Win32_OperatingSystem',
                    wbemFlagReturnImmediately or
                    wbemQueryFlagShallow, nil);

    // Replicate VBScript's "for each" construct
    Enum := (wmiObjectSet._NewEnum) as
                     IEnumVariant;
    while (Enum.Next(1, ovVar, lwValue)=S_OK) do
    begin
      wmiObject := IUnknown(ovVar) as SWBemObject;
      propSet   := wmiObject.Properties_;
      propEnum  := (propSet._NewEnum) as
                                 IEnumVariant;

      // Replicate VBScript's "for each"
      // construct
      while (propEnum.Next(1,ovVar,lwValue) = S_OK) do
      begin
        wmiProp := IUnknown(ovVar) as SWBemProperty;
        sValue := '';
        if VarIsNull(wmiProp.Get_Value) then
          sValue := 'Is Null'
        else
        case wmiProp.CIMType of
          wbemCimtypeSint8,
          wbemCimtypeUint8,
          wbemCimtypeSint16,
          wbemCimtypeUint16,
          wbemCimtypeSint32,
          wbemCimtypeUint32,
          wbemCimtypeSint64:
            if VarIsArray(wmiProp.Get_Value) then
            begin
              if VarArrayHighBound
                   (wmiProp.Get_Value, 1) > 0 then
                for Count := 1 to
        VarArrayHighBound(wmiProp.Get_Value, 1) do
                  sValue := sValue + ' ' +
                IntToStr(wmiProp.Get_Value[Count]);
            end
            else
              sValue:= IntToStr(wmiProp.Get_Value);

          wbemCimtypeReal32,
          wbemCimtypeReal64:
            sValue := FloatToStr(wmiProp.Get_Value);

          wbemCimtypeBoolean:
            if wmiProp.Get_Value then
              sValue := 'True'
            else
              sValue := 'False';

          wbemCimtypeString,
          wbemCimtypeUint64:
            if VarIsArray(wmiProp.Get_Value) then
            begin
              if
    VarArrayHighBound(wmiProp.Get_Value,1)> 0 then
                for Count := 1 to
        VarArrayHighBound(wmiProp.Get_Value, 1) do
                  sValue := sValue + ' ' +
                         wmiProp.Get_Value[Count];
            end
            else
              sValue :=  wmiProp.Get_Value;

          wbemCimtypeDatetime:
            sValue :=  wmiProp.Get_Value;
        end;
        mWMI.Lines.Add(wmiProp.Name + ' = ' +
                                         sValue);
      end;
    end;
  finally
    wmiLocator.Free;
  end;
end;
```

Listing 9 – Iterating over the WMI properties

Figure 3 presents a screenshot of a small application that we can use to try out our WMI code. Listing 9 is attached to the Advanced WMI Class Info button's OnClick event. Incidentally, listing 8 is attached to the Simple WMI Class Info button's OnClick event, so you can try out that code too.

Listing 9 iterates over the [single] wmiObject's **Properties_** property, extracting each property and then adding the property's details to a TMemo. As figure 3 demonstrates, it is possible to replicate the VBScript's functionality, albeit we have to iterate over all the individual properties. At least now we can seek out specific WMI class properties, such as the Win32_OperatingSystem Caption property, or the CSDVersion (service pack). How many of your clients/customers just think they are running "Windows", not knowing which version (95,98,Me, 2000, XP, etc.) How many of your clients/customers know which service pack is installed?

Figure 3

Using WQL

In the first part of this article, I mentioned the fact that WMI uses a repository or database to hold the WMI class information – the CIM Repository. We know that databases typically have a query language. The CIM Repository is no different – it supports the Windows Management Instrumentation (WMI) Query Language (WQL).

Consider listing 10a – it should be familiar to us, listings 8 and 9 used this line of code to obtain an instance of the Win32_OperatingSystem WMI class.

```
// Obtain an instance of the WMI class
wmiObjectSet := wmiServices.InstancesOf(
                    'Win32_OperatingSystem',
                    wbemFlagReturnImmediately or
                    wbemQueryFlagShallow, nil);
```

Listing 10a – Obtaining an instance of Win32_OperatingSystem

Earlier in this article I moaned about how we had to iterate over all the Properties in the Properties_ collection. Thankfully WQL can stop me moaning! WQL provides us with a means of choosing the properties that we would like to appear in the Properties_ collection. Listing 10b demonstrates how a few changes can extract just the Caption and CSDVersion properties.

```
// Obtain an instance of the WMI class
wmiObjectSet := wmiServices.ExecQuery(
     'SELECT Caption,CSDVersion FROM ' +
     edtClass.Text, 'WQL',
     wbemFlagReturnImmediately, nil);
```

Listing 10b – WQL in action

If we augment the Simple WMI Class Info button's OnClick event to use the code presented in listing 10b, clicking on that button produces figure 4. Notice that the WQL statement specifically asked for Caption and CSDVersion, yet the Properties_ collection also contains the Name property. The Name property is a WMI "key" type: as such the WQL engine will return the key types as if they were specified in the WQL SELECT clause.

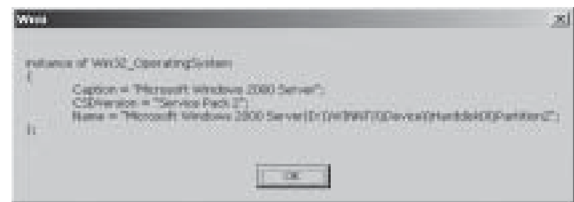

Figure 4

Connecting To Remote Computers

Using WMI to connect to remote computers moves us into a realm where security plays an important role. Firstly, we must have the necessary rights to access the remote computer – we need a valid user name and a password. Secondly, our application (in this case wmi.exe) has to be firewall-friendly – I use ZoneAlarm on my local computer, and Norton Internet Security on the remote computer. The local computer's firewall has to allow wmi.exe (the WMI test application) requests to pass through to the remote computer. In turn the remote computer has to be able to receive the requests. In this example, both computers need to be configured accordingly – obviously in your environment your mileage may vary.

If you recall listing 2, it described the ConnectServer method that is used to access the WMI classes. ConnectServer took a number of parameters, including strUser and strPassword. By default, the WMI test application uses the local computer, as specified by the '.' in the Computer edit control. However, the WMI test application has a couple of edit controls that allow us to specify the User Name and the Password.

On my small network the local computer is running Windows 2000 Server SP2 and my remote laptop is running Windows 2000 Professional SP2. My local computer is known as CAMW2K and my laptop is called DEVCAM. If we enter the necessary computer name, user name and password into the WMI test application (running on the local CAMW2K computer), then click on the Advanced WMI Class Info button, we magically receive information from DEVCAM (which could be anywhere on our network). Figure 5 demonstrates the output from the WMI test application.

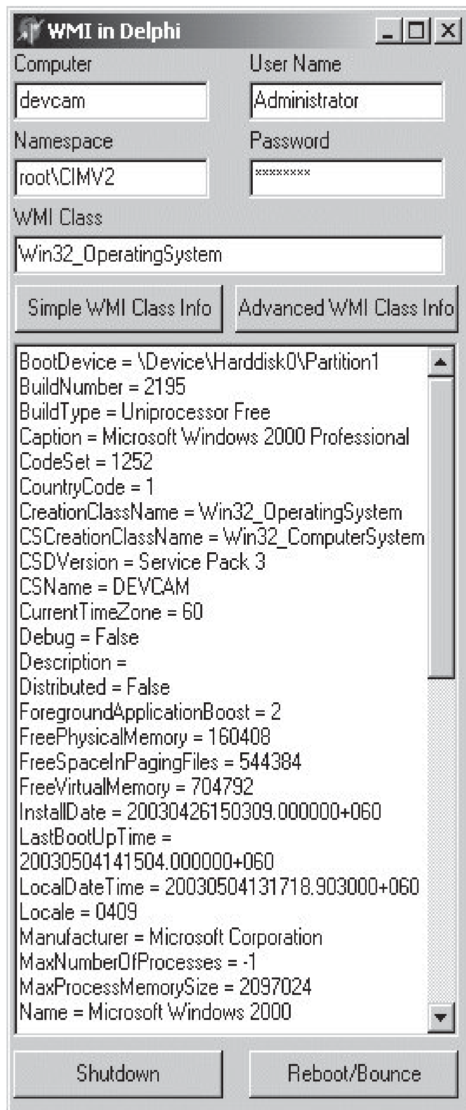

Figure 5 – Using WMI on remote computers

Bouncing Remote Computers

The astute amongst you will have noticed that there are two buttons at the bottom of the WMI test application: Shutdown and Reboot/Bounce. Clicking on these buttons whilst Computer is ‘.’ has the obvious effect of shutting down or rebooting the local PC…save your work before you try this!

A very practical use of WMI is the ability to ‘bounce’ or reboot remote PCs. How often have you needed to reboot a computer elsewhere on your network? Perhaps the computer is in a locked room, or perhaps the computer is a long walk from where you are sitting? Listing 11 presents the code required to use WMI in such a way that we can force the local or remote computers to reboot themselves. Remember to change the username and password if you plan to try this code for yourself. (Funnily enough, my laptop’s Administrator password is not RICHPLUM!).

```
procedure TfrmMain.btnRebootClick(Sender:
                     TObject);
var
  wmiLocator:   TSWbemLocator;
  wmiServices:  SWbemServices;
  wmiObjectSet: SWbemObjectSet;
  wmiInst,
  wmiParams,
  wmiObject:    SWbemObject;

  Enum:         IEnumVariant;
  propValue,
  ovVar:        OleVariant;
  lwValue:      LongWord;
  wmiMethod:    SWbemMethod;
  wmiProperty:  SWbemProperty;

begin
  wmiLocator  := TSWbemLocator.Create(self);
  wmiServices := wmiLocator.ConnectServer(
                   ‘devcam’,
                   ‘root\cimv2’,
                   ‘Administrator’,
                   ‘RICHPLUM’, ‘’,
                   ‘’, 0, nil);

  wmiServices.Security_.Privileges.Add(
               wbemPrivilegeShutdown, True);

  wmiObjectSet := wmiServices.ExecQuery(
    ‘SELECT * FROM Win32_OperatingSystem ’ +
    ‘WHERE Primary=True’,
    ‘WQL’, wbemFlagReturnImmediately, nil);

  Enum := (wmiObjectSet._NewEnum) as
                              IEnumVariant;
  while (Enum.Next(1, ovVar, lwValue)=S_OK) do
  begin
    wmiObject   := IUnknown(ovVar) as
                                SWBemObject;
    wmiMethod   :=
     wmiObject.Methods_.Item(‘Win32Shutdown’,0);
    wmiParams   := wmiMethod.InParameters;
    wmiInst     := wmiParams.SpawnInstance_(0);
    wmiProperty :=
              wmiInst.Properties_.Add(‘Flags’,
              wbemCimtypeSint32, False, 0);
    propValue   := EWX_REBOOT;
    wmiProperty.Set_Value(propValue);

    wmiObject.ExecMethod_(‘Win32Shutdown’,
                         wmiInst, 0, nil);
  end;

  wmiLocator.Free;
end;
```

Listing 11 – Rebooting remote computers

I suppose this feature could also be construed as a humorous [April 1st?] prank in the right environment. Not that you would catch *me* doing something like that...no sir.

Summary

Over the course of these two articles I have covered more WMI than I needed to solve my original problem. However, I have learnt that WMI is a powerful tool (and in the wrong hands, listing 11, is a dangerous tool!), and that there is a lot more to WMI than I have covered. Hopefully these two articles will encourage you to explore WMI in more detail and ideally to incorporate it into your own applications.

Finally, I do hope these two articles made sense: if they did, then you can safely say "you're WMI".

Resources

Iterating over a collection – Mar 30 1998 – DIL

More Automation in Delphi Session by Brian Long – Oct13 2000 – DIL

MSDN WMI Downloads: http://msdn.microsoft.com/library/default.asp?url=/downloads/list/wmi.asp

Craig is an author, developer, speaker and Dilbert Evangelist – he specialises in all things XML, particularly SOAP and XSLT. He can be reached via e-mail at: Craig@isleofjura.demon.co.uk, or via his web site: http://www.craigmurphy.com (where you can also find the source code and PowerPoint files for all of Craig's UK-BUG articles and presentations).