

Instrumentation in C# (Part 1)

by Craig Murphy

If you have been reading my recent articles elsewhere, you will know that I have been ranting on about something called WMI – Windows® Management Instrumentation. WMI is a means of programmatically taking control of various parts of the computer. We can use WMI to extract all kinds of information about the operating system, the disc drives, the printers, etc. that are attached to a computer. We can even reboot the computer if required. The beauty of WMI shines through when we realise that we can do all of this, not just from a local computer, but also from another computer elsewhere on the network. In a nutshell, WMI gives you the ability to administer your computer and other computers on your network.

Don't Be Afraid

Prior to the arrival of .NET, WMI may have been one those topics that you steered clear of for a variety of reasons, possibly fear. Your fear may have been justified because, prior to .NET, WMI was COM-based, was often demonstrated using VBScript and had some reliance on late binding. So who could blame you? Thankfully, as with most things in .NET, all this is has changed. For a start, we can use C# in place of VBScript (phew!)

And do not be afraid if you have not got Visual Studio.net (VS.NET) – all of the examples in this article have been developed using VS.NET and have been tested using SharpDevelop. In fact, the code download associated with this article (www.richplum.net/downloads/default.asp) is available in two flavours: VS.NET and SharpDevelop.

What is WMI?

WMI is a mechanism that allows programmatic access to virtually all of the Windows resources you could possibly want to work with. The Windows resources that I'm talking about include such things as system information, BIOS information, boot configuration, installed codecs, drive partition information, environment variables, operating system information – the list goes on. Microsoft has produced a chart of the WMI classes that are provided – it fits onto a sheet of A3...and covers less than 20% of the WMI classes that are available. If you would like to see this chart, search the Microsoft Developer Network (MSDN) web site for 'wmichart'.

If you examine the task list on your computer, you should find a process called WinMgmt.exe – it is this process that is the core of WMI. WinMgmt.exe is known as the Computer Information Model Object Manager (CIMOM).

The CIMOM maintains a database, or repository, that contains object definitions, class and instance definitions that can be used to access and maintain system configuration information. The CIMOM database has to be populated – this is achieved by Managed Resources, such as Windows itself, a Windows service, or even an application, providing CIMOM with information about the API that it would like WMI to be aware of.

Sitting in between the Managed Resource and the CIMOM is the WMI Provider. The WMI Provider is essentially a marshalling layer that provides a consistent interface between the Managed Resource and the CIMOM. This article will be concentrating on using WMI (consuming), rather than building WMI-compliant applications – hence we will not go in to much detail as to how this interface actually works.

The final part of the equation is something that uses the WMI classes – a WMI consumer. A WMI consumer can take many forms – it could be an ASP page, a Visual Basic Script (VBS), or a Delphi application. If you have ever run a software tool that audits your PC, it is very likely that used WMI classes to access system information – in which case the audit software is considered a WMI consumer. Equally, the “System Info...” button found in Help/About Microsoft Word (for example) is a WMI consumer. Figure 1 presents the Elementary WMI Architecture.

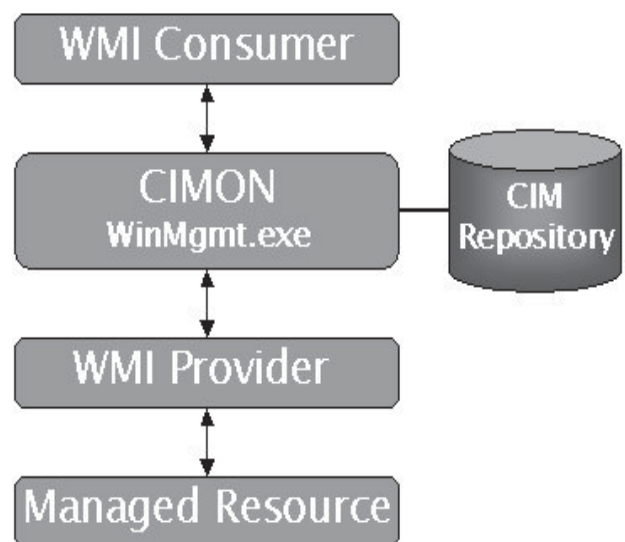


Figure 1 – Elementary WMI Architecture

WMI – A Simple Example

I will apologise now – this is going to be a Visual Basic Script (VBScript) example. WMI classes can be consumed using scripting languages (via the Scripting API for WMI) and via the COM API for WMI. Using the COM API for WMI requires us to write a reasonable amount of code – and that is something that we will do in the second article. However, with so many of us becoming multi-lingual, and the relative simplicity of VBScript, the examples should not present you with any problems.

As most developers will agree, when we ask an end user the question “What OS are you using?” the replies we receive range from precise to generic...from Windows 2000 SP2 down to just Windows. The generic response is

not much good if you are trying to work out why your application refuses to work on their machine! So, as Microsoft has demonstrated, including some system information in your application's Help/About dialog can be a useful time saver.

There are many ways of obtaining this information, the most obvious being to trawl the Windows Registry. This would work, however the format of the registry settings varies from Windows 95 to Windows XP. Tapping into the registry for information is not always that easy - network card information is stored via the use of linked GUIDs. Incorporating GUID tracing code into your application just adds to the complexity. To avoid these problems, we can make use of Microsoft's recommended WMI to use a standard, consistent interface that allows us to manage the entire machine's information/resources regardless of which OS is installed.

We have already seen that WMI is based around the concept of providers. A core provider is known as the *Win32 provider*, and is implemented by *cimwin32.dll*. The Win32 provider is able to supply information about the computer, disks, peripheral devices, files, folders, file systems, networking components, operating system, printers, processes, security, services, shares, and much more.

There are literally hundreds of WMI classes available – these are organised into groups that are identified using namespaces. The Win32 provider's classes are found in the *root\cimv2* namespace – which includes the classes that provide information about the computer and installed OS. As you might expect, there is the concept of a default namespace. The default namespace is typically *root\cimv2*, which is why it can be omitted. However, for the sake of completeness, Listing 1 demonstrates how to use WMI with or without a namespace.

Listing 1 presents the Visual Basic Script required to extract a variety of OS-specific facets. Most of the facets (ServicePackMajorVersion, OSType, etc.) are self-explanatory.

```

` 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 objOperatingSystem In colOperatingSystems
  Wscript.Echo "Name: " &
    objOperatingSystem.Name &
    vbCrLf & "Caption: " &
    objOperatingSystem.Caption &
    vbCrLf & "CurrentTimeZone: " &
    objOperatingSystem.CurrentTimeZone &
    vbCrLf & "LastBootUpTime: " &
    objOperatingSystem.LastBootUpTime &
    vbCrLf & "LocalDateTime: " &
    objOperatingSystem.LocalDateTime &
    vbCrLf & "Locale: " &
    objOperatingSystem.Locale &
    vbCrLf & "Manufacturer: " &

```

```

objOperatingSystem.Manufacturer &
vbCrLf & "OSType: " &
objOperatingSystem.OSType &
vbCrLf & "Version: " &
objOperatingSystem.Version &
vbCrLf & "Service Pack: " &
objOperatingSystem.ServicePackMajorVersion &
"." &
objOperatingSystem.ServicePackMinorVersion &
vbCrLf & "Windows Directory: " &
objOperatingSystem.WindowsDirectoryNext

```

Listing 1 – Extracting OS info using WMI

Assuming that you execute Listing1.vbs by double clicking on it, Figure 2 presents the output. Alternatively, if you wish to execute Listing1.vbs via the command-line, simply entering *cscript Listing1.vbs* at the command-line will produce similar results.

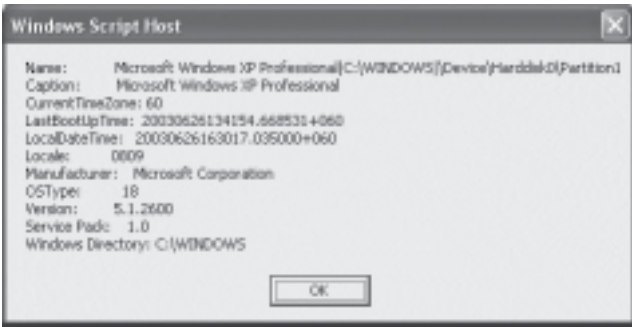


Figure 2 – VBScript has some uses!

Enough reminiscing! Let's get on with looking at WMI in .NET.

WMI the .NET way

If we are to use WMI in .NET, there are two things we must do before we begin. Firstly, we must add a reference to the System.Management namespace. If you are using VS.NET, this is a fairly easy task: simply right-click on the References node of the Solution Explorer then click on the Add Reference menu option. Using the .NET tab, scroll down until you find System.Management, as shown in Figure 3. Left clicking on Select adds the reference to your project. Finally, click on OK to return to the project workspace.

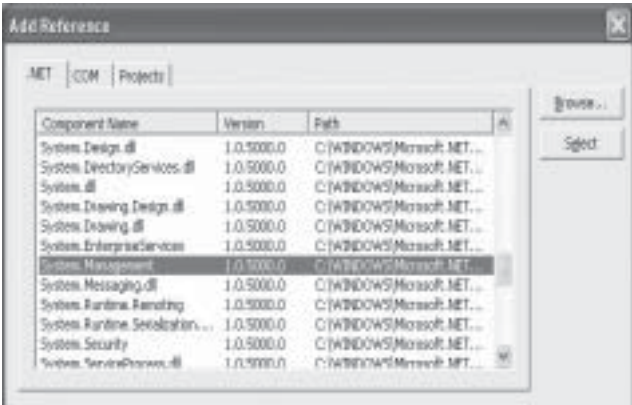


Figure 3 – Adding a reference to System.Management

Secondly, we need to create a class wrapper for the WMI class that we are going to use. Creating a class wrapper will allow us to take advantage of early binding, which is a good thing. Whilst late binding (i.e. at run-time) works, it requires extensive testing to ensure that every late-bound method call/property access does not generate a run-time error. It is possible to use WMI in .NET without creating a class wrapper, however I prefer early binding over late binding. Let's take a look at how we can create a class wrapper using a .NET helper application.

Early Binding

One of the .NET helper applications will create a class wrapper around the WMI classes (in a similar fashion to Delphi's web service interface-based proxies). This will provide design-time access to the WMI class and all the properties that are associated with it – most notably it will allow us to make use of code completion within our chosen IDE.

The .NET helper application is called **mgmtclassgen**. Some of the current .NET documentation refers to it as "managementclassgen", which does not exist, so watch out for that.

If you are using VS.NET, mgmtclassgen can be run from a VS.NET Command Prompt (it sets up environment variables allowing mgmtclassgen.exe to be found without the need to specify the full path). The basic syntax for mgmtclassgen is:

```
mgmtclassgen Win32_OperatingSystem /L cs
```

mgmtclassgen can be used to create class wrappers for use in a C# (cs) environment, a JavaScript (js) environment, and, for the sake of completeness I suppose, for use in a VB.NET (vb) environment. The '/L' parameter is used to specify the language that is required. There are other command-line parameters available: WMI can be used to work with local and remote computers, for this reason there are command-line options that allow us to provide usernames and passwords for the mgmtclassgen application to use. The '/L' parameter is all that we need in order to get our first .NET WMI application up and running.

So, having executed mgmtclassgen, it creates a class wrapper in OperatingSystem.cs for us. OperatingSystem.cs is some 2098 lines in length! Before we move on, it's probably worth taking a look at OperatingSystem.cs – I've included some of the more salient points in Listing 2 (at which point our Editor is relieved to see that I've not included all 2098 lines!).

```
namespace ROOT.CIMV2.Win32
<snip>
public class OperatingSystem :
System.ComponentModel.Component {

[Browsable(true)]
[DesignerSerializationVisibility
(DesignerSerializationVisibility.Hidden)]
[Description("The Caption property is a short
textual description (one-line string) of
the object.")]
public string Caption {
get {
return ((string)(curObj["Caption"]));
}
}
}
<snip>
```

Listing 2 – Abbreviated OperatingSystem.cs

Listing 2 defines the **ROOT.CIMV2.Win32** namespace. By default, mgmtclassgen will create the class wrapper in this namespace. Using command-line parameters, it is possible to specify your own namespace if required. Each time mgmtclassgen is used to create a class wrapper for the WMI Win32 classes, a new [C#] source code file is created. Each individual C# file will use the same namespace, ROOT.CIMV2.Win32, which is perfectly acceptable. Multiple class wrappers can share the same namespace but do not have to be in the same source code file.

Listing 2 also defines a class OperatingSystem that contains many properties relating to the WMI properties for Win32_OperatingSystem. I have highlighted the Caption property – in the WMI class, Win32_OperatingSystem, the caption property contains the name of the operating system, in my case Microsoft Windows 2000 Professional. All other WMI class properties are defined in a similar fashion.

The last aspect of Listing 2 that is worthy of attention is the inclusion the [Description...] attributes. The class wrapper not only includes all the WMI class properties, but it also includes the associated help string that goes with each property. .NET has some great support for decorating classes and class properties using attributes – we'll see how to make use of the Description attribute later in this article.

Using The Class Wrapper

Before we can use OperatingSystem.cs, we need to move it into our WMI project directory – this assumes that you have created a new/empty project in your chosen IDE. Once you have created a new/empty WinForms project, add the OperatingSystem.cs to your project. If you are using SharpDevelop, remember to include OperatingSystem.cs in your list of files to be compiled (Project Options → Compile), otherwise there will be compile-time errors. Alternatively, you can right-click on OperatingSystem.cs, then choose Include → Compile.

We need something to display our WMI class information in, so drop a ListBox control on the main form. Next, we need to drop a Button control – it is a very basic example but it is enough to demonstrate the key points. I called the Button control btnFetchOSInfo – Listing 3 presents the code that is attached to the button's Click event. However, before this code will compile, we need to add a reference to OperatingSystem.cs – at the top of your main form's source file, add the following statement:

```
using ROOT.CIMV2.Win32;

private void btnFetchOSInfo_Click(object
sender, System.EventArgs e)
{
foreach( ROOT.CIMV2.Win32.OperatingSystem OS
in
ROOT.CIMV2.Win32.OperatingSystem.GetInstances())
{
lbWMI.Items.Add(OS.Caption);
lbWMI.Items.Add(OS.CSDVersion);
}
}
```

Listing 3 – Using OperatingSystem.cs

If all goes well, the example should now compile. If you are typing the example code in, remember that C# is case-sensitive, so lbWMI is different from lbwmi, for example. Assuming that everything compiled, run the example and then click on the “Fetch OS info” Button – if all went well, Figure 4 should be familiar.

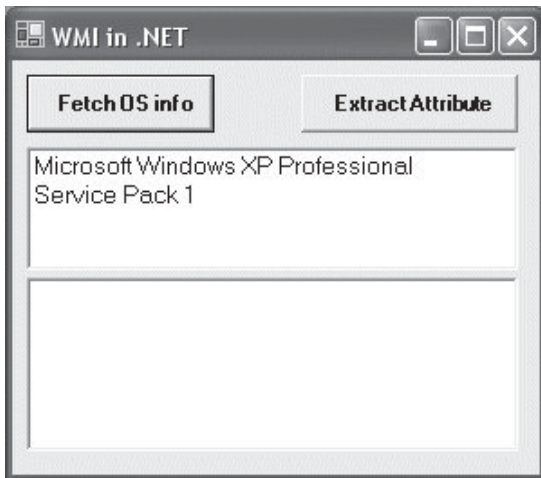


Figure 4 – using OperatingSystem.cs

Using the Description Attribute

Although attributes are not a WMI specific item, they do make up most of the OperatingSystem.cs source code file that mgmtclassgen created, so they are worthy of a brief discussion here. The OperatingSystem class is augmented or decorated with a number of [Description...] attributes – if you look at Listing 2 again, you will see the attribute contains the WMI help text for the Caption property. How can we extract that help text for use in our applications?

To demonstrate extracting the [Description...] attribute contents, add another Button and a TextBox to the main form. I called the Button ‘btnExtract’ and gave it a caption of ‘Extract Attribute’ – Listing 4 provides the code for the Click event.

```
private void btnExtract_Click(object sender,
    System.EventArgs e)
{
    txtWMI.Clear();

    foreach(ROOT.CIMV2.Win32.OperatingSystem OS in
        ROOT.CIMV2.Win32.OperatingSystem.GetInstances())
    {
        AttributeCollection attributes =
            TypeDescriptor.GetProperties(OS)
                ["Caption"].Attributes;

        DescriptionAttribute myAttribute =
            (DescriptionAttribute)attributes
                [typeof(DescriptionAttribute)];

        txtWMI.Text = myAttribute.Description;
    }
}
```

Listing 4 – Extracting Attributes

Once again, if the code compiles, run the example and then click on the ‘Extract Attribute’ button – if all goes well, Figure 5 shouldn’t look too unfamiliar.

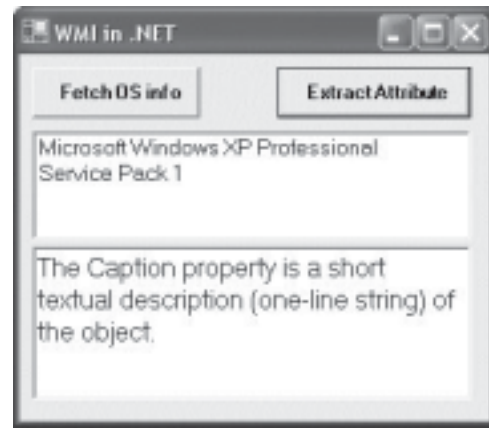


Figure 5 - Extracting attributes.

Granted, the [Description...] attribute for the Caption property is perhaps not the best help-string that you could imagine, but it is a useful demonstration. If you skim over OperatingSystem.cs you will find more meaningful [Description...] attributes.

WMI Administrative Tools

If this article has captured your attention and you would like to explore WMI in more detail, the WMI Administrative Tools will prove invaluable. In the Resources section of this article I have provided a link to the MSDN web site – most of the non-.NET WMI downloads are available from this link, including the Windows ME, 98 and 95 WMI installations. If you are interested in learning more about WMI in .NET, I can recommend examining the System.Management and System.Management.Instrumentation namespaces in more detail.

The WMI Admin Tools is a 4.5mb download from the MSDN web site, so even in a “broadband denied” location it is not a painful download. If you have downloaded and installed the WMI Admin Tools, you will have noticed a new program group has been created in the Programs group – WMI Tools. The WMI Admin Tools consist of four tools, as shown in figure 6. The primary tool that we can use to help us understand WMI is the WMI CIM Studio. CIM Studio is a graphical front-end that allows us to navigate the WMI classes on both local and remote computers.



Figure 6 – The WMI Administration Tools

The WMI Admin Tools do not require the .NET framework. VS.NET incorporates access to the Management Classes via Server Explorer. The Server Explorer interface to the Management Classes actually exposes the mgmtclassgen functionality such that it is possible to create the WMI class wrappers within the IDE.

Resources

SharpDevelop, free C# IDE: <http://www.icsharpcode.net/>
 MSDN WMI Downloads: <http://msdn.microsoft.com/library/default.asp?url=/downloads/list/wmi.asp>

What's Next?

Apart from the follow-up to this article, I've been playing with XML in .NET, so don't be too surprised if that's the subject of a future article!



Craig is an author, developer, speaker and Dilbert Evangelist – he specialises in all things XML, particularly SOAP and XSLT. Craig is also preparing to become evangelical about C#, Test-Driven Development and Extreme Programming. He can be reached via e-mail at: Craig@isleofjura.demon.co.uk,

or via his web site: <http://www.craigmurphy.com>).

We were very lucky to get this article. Craig wrote it sitting in Bristol airport on a lonely Saturday morning, waiting for the first flight back to Edinburgh, with his laptop precariously plugged in 12 feet across the corridor, and acting as a skipping-rope by the cleaners. He is now enjoying a well-earned holiday.